IBM

B is for BASIC
An Introduction to IBM BASIC

Program Product

B B B B B

A A A A A

S S S S S

I I I I I

C C C C C

Order Number:
GC26-4102-0

Program Numbers:
5668-996 (VM)
5665-948 (MVS)

Release Number:
2

IBM

B is for BASIC
An Introduction to IBM BASIC

Program Product

Order Number:
GC26-4102-0

Program Numbers:
5668-996 (VM)
5665-948 (MVS)

Release Number:
2

First Edition (February 1986)

This edition applies to Release 2 of IBM BASIC, Program Products 5668-996 (VM) and 5665-948 (MVS), and to any subsequent releases until otherwise indicated in new editions or technical newsletters. This edition incorporates much of the information in *B is for BASIC An Introduction to VS BASIC under TSO*, SC28-8300 and *B is for BASIC An Introduction to VS BASIC under CMS*, SC28-8310.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, and 4300 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

Preface

The intention of this book is to familiarize you with one of the many ways a computer can be used to solve problems.

When you use a computer, you have to be able to communicate with it in a language it understands, so you can tell it what you want it to do. There are many languages computers can understand. BASIC (which stands for Beginners' All-purpose Symbolic Instruction Code) is one that is particularly versatile and easy to use. Different forms of it run on all sizes and types of computers, from small personal computers to large mainframes. This book describes one of the varieties of BASIC, called IBM BASIC,[1] that operates in the VM (Virtual Machine) and MVS (Multiple Virtual Storage) operating environments. The information in this book is system independent; that is, it applies to both operating environments in which IBM BASIC runs.

This book describes how to use BASIC to solve simple problems. It does not describe the entire language, but it should be enough to get you started as a programmer. Later on, if you want to learn about the full range of things you can do with the language, you can read about more advanced topics in programming in *IBM BASIC Programming Guide, SC26-4027*, and about the complete BASIC language in *IBM BASIC Language Reference, GC26-4026*.

IBM BASIC operates with two kinds of terminals, full-screen terminals (such as those in the IBM 327X family) and line-oriented terminals. BASIC can accommodate both kinds, but the examples in this book assume you're using a full-screen terminal.

There will be many examples as you go along. If possible, sit at a terminal while you read this book, and type in all of the examples. If you can't use this book directly at a terminal, try to get to one often to try out what you've read in each chapter. Pictures showing what your terminal screen should look like appear in the margins of the pages in chapters 1 and 2. Read the explanations and then look at the pictures of the screens as you type.

A summary of BASIC statements and commands is included at the end of each chapter. Chapter 12, "Command and Statement Summary" on page 79 contains a complete summary of all statements and commands presented in the book.

There are exercises after each chapter. Do the exercises, and make up some of your own, until you feel comfortable with the material in that chapter. Then go on to the next chapter. The answers to the exercises are in Appendix A, "Answers to Exercises" on page 81.

We encourage you to experiment freely at your terminal! Pretty soon you'll feel like an old hand at using BASIC.

Prerequisite Knowledge

We do *not* assume you have had previous experience with either BASIC or any other programming language. However, before you begin to read this book and try out exercises at your terminal, you should be familiar with the terminal you are using and its keyboard. You should also know how to log on to your computer, and something about your operating environment. Ask a friend across the hall or your system administrator to spend 15 minutes to an hour showing you the fundamentals.

---

1 When we talk about BASIC in this book, we always mean IBM BASIC.

Contents

Chapter 1. Getting Started

Before you sit down at the terminal and face your terminal screen, let's look at what's behind that screen. The computer. You don't have to understand it at the "nuts and bolts" level. All you have to learn is how to make it do the things you want it to do.

The computer simply follows instructions. And it follows them to every dot of an *i*, every cross of a *t*. But it refuses your request when you tell it to do something that some other programmer (a system programmer who makes it possible for you to give the computer instructions in BASIC) has already determined impossible — such as dividing by zero.

What you will be doing is giving the computer instructions. And, as we will show you, these instructions must be free of errors, because the computer has no "common sense" and can't understand you when you make mistakes.

Connecting With The Computer

It's time to begin your first session. Turn on your terminal and log on to the computer.

Note: As explained in the preface, you need to know how to log on to your computer (as well as the fundamentals of the terminal keyboard) to use this book.

The operating system you're using is called CMS (short for Conversational Monitor System), if you're in the VM operating environment, or TSO (short for Time Sharing Option) if you're in the MVS operating environment.

When you're logged on, you'll get a message that indicates the computer is ready for you to type instructions. In CMS it may look something like this:

```
R; T=14.32/22.73 14:59:28
```

and in TSO, like this:

```
READY
```

Connecting with IBM BASIC — CMS

you enter ▶
```
R; T=14.32/22.73 14:59:28
basic
```

Here's what happens:

```
IBM BASIC/VM VERSION 1 RELEASE 2.0  1985/07/25  16:27
   (c) Copyright IBM Corporation  1982, 1985
* _
```

IBM BASIC is ready for you to start typing here.

Connecting with IBM BASIC — TSO

you enter ▶
```
READY
basic
```

Here's what happens:

```
IBM BASIC/MVS VERSION 1 RELEASE 2.0  1985/07/25 16:27
   (c) Copyright IBM Corporation  1982, 1985
* _
```

IBM BASIC is ready for you to start typing here.

Connecting With IBM BASIC — The BASIC Command

After you're logged on, you must specify that you want the services of IBM BASIC. Just type:

```
basic
```

and press the ENTER key. (You always need to remember to press the ENTER key after typing a command.)

Note: In this book, we'll always show what you should enter at your terminal in blue.

Typing the BASIC command is the standard way to enter IBM BASIC; however, your installation might have created a different command for you to use. Check with your system administrator to be sure.

BASIC lets you know it's ready to start work by answering something like this, if you're using the VM (CMS) version of BASIC:

```
IBM BASIC/VM VERSION 1 RELEASE 2.0  1985/07/25  16:27
   (c) Copyright IBM Corporation  1982, 1985
* _
```

And this, if you're using the MVS (TSO:) version of BASIC:

```
IBM BASIC/MVS VERSION 1 RELEASE 2.0  1985/07/25  16:27
   (c) Copyright IBM Corporation  1982, 1985
* _
```

What If Your Screen Looks Different?: If your entry line (* _) is at the *top* of your screen instead of at the *bottom*, type the command SET TERM SCROLL on the entry line, and press ENTER. Your screen will then look like those shown in this book.

Back to the BASIC Screen: The first line identifies the version and release of IBM BASIC you're using, and the date and time. The * _ identifies the line on which you'll enter BASIC commands or statements. It's called the **entry line, prompt line,** or **input area.** As you type, each character is immediately displayed on that line, after the asterisk, The _ is the **cursor,** which shows you where characters you type will appear on your terminal screen. Each time you complete a line of instructions to BASIC, you press the ENTER key to tell the computer that the line is complete.

This procedure — logging on and then specifying IBM BASIC — begins every BASIC session.

Leaving IBM BASIC

you enter ▶

```
IBM BASIC/XX VERSION 1 RELEASE 2.0  1985/07/25  16:27
   (c) Copyright IBM Corporation  1982, 1985
* quit
```

Here's what happens in CMS:

```
R; T=14.32/22.73 14:59:28
```

Here's what happens in TSO:

```
READY
```

Leaving IBM BASIC — The QUIT Command

Whenever you want to leave the BASIC environment, just type the following command and press ENTER:

```
* quit
```

Your BASIC session will be ended immediately, and you'll find yourself back under the control of your operating system (you'll get the ready message — R; or READY). At that point, you can go on and do other work at your terminal, or log off the system.

If you'd like to try it, quit now and then reenter the BASIC command to get back into BASIC.

If All Else Fails...

We've tried to structure this book so you can follow the instructions step-by-step and not run into trouble, but events outside our knowledge or control might get you into a situation in which your screen doesn't look like the book says it should, and you might be confused about what to do next.

If that happens, press ENTER, if necessary, to try to get the BASIC prompt (* _), and then start over with whatever you were doing. If you can't get the BASIC prompt, or you're *still* confused, find the friend across the hall we mentioned in the preface to this book (the one who showed you how to log on and to use the terminal): you probably need help from an expert.

Using Immediate Statements

you enter ▶

```
IBM BASIC/XX VERSION 1 RELEASE 2.0  1985/07/25  16:27
   (c) Copyright IBM Corporation  1982, 1985
 * print 'hello'
```

Here's what happens:

```
IBM BASIC/XX VERSION 1 RELEASE 2.0  1985/07/25  16:27
   (c) Copyright IBM Corporation  1982, 1985
 * print 'hello'
hello
 * _
```

Getting Results Immediately — Immediate Statements

For your first instruction to BASIC, let's use one of the simplest — an **immediate statement**, which means just that. When you type an immediate statement and press the ENTER key, BASIC does what you've told it to do right away.

One type of immediate statement instructs BASIC to display a specific line of text, like a word or phrase. For example, enter the following statement:

```
* print 'hello'
```

In the PRINT statement, you must always use quotes around any characters (except numbers) that are to be printed out literally as you entered them. You can, however, use either apostrophes or (double) quotation marks. In other words:

```
print "hello"
```

is the same as:

```
print 'hello'
```

Many BASIC programmers prefer apostrophes, because you don't have to shift to type them.[2]
In programming terms, we call such groups of characters **character constants** or **string constants**.[3]

Now when you press the ENTER key, you'll see the following on your screen:

```
 * print 'hello'
hello
 * _
```

BASIC printed out what you told it to. And note that, when BASIC returns with the asterisk and cursor to prompt you for another instruction, all the previous lines move up on your screen. This is called **scrolling** and enables you to monitor what you've entered and how BASIC has responded, providing you a history of what has occurred.

---

2 However, you might not always have a choice about which kind of quote to use. If you wanted BASIC to type something with an apostrophe, like `It's spring!`, you'd have to type it like this:

```
print "It's spring!"
```

3 Examples in this book show BASIC commands and statements that you enter at the terminal in lowercase letters (that is, `print` as opposed to uppercase — `PRINT` — or mixed case — `Print` — because most BASIC programmers prefer to type without having to shift. IBM BASIC interprets commands and statements the same regardless of case. However, if you want BASIC to print out a character string for you in mixed case, then type it in in mixed case — for example, `print 'Hello'`.

Making an Error

```
IBM BASIC/XX VERSION 1 RELEASE 2.0  1985/07/25  16:27
   (c) Copyright IBM Corporation  1982, 1985
 * print 'hello'
hello
 * pzint 'hello'
```

you enter ▶

Here's what happens:

```
IBM BASIC/XX VERSION 1 RELEASE 2.0  1985/07/25  16:27
   (c) Copyright IBM Corporation  1982, 1985
 * print 'hello'
hello
 * pzint 'hello'
        1
###1) BAS30013S PRECEDING IDENTIFIER IS NEITHER A
      COMMAND NOR A STATEMENT KEYWORD. '=' '(' OR
      ',' EXPECTED FOR ASSIGNMENT STATEMENT.
 *
  _
```

Correcting an Error

```
IBM BASIC/XX VERSION 1 RELEASE 2.0  1985/07/25  16:27
   (c) Copyright IBM Corporation  1982, 1985
 * print 'hello'
hello
 * pzint 'hello'
        1
###1) BAS30013S PRECEDING IDENTIFIER IS NEITHER A
      COMMAND NOR A STATEMENT KEYWORD. '=' '(' OR
      ',' EXPECTED FOR ASSIGNMENT STATEMENT.
 * print 'hello'
```

Simply reenter the line correctly.

What Happens If You Make An Error

Now we'd better show you how precise you must be in entering commands in BASIC. Try entering:

```
 * pzint 'hello'
```

Maybe a friend of yours, reading the screen over your shoulder, would understand that you hit the "Z" key instead of the "R." But BASIC is just puzzled, and sends you the following message:

```
 * pzint 'hello'
        1
###1) BAS30013S PRECEDING IDENTIFIER IS NEITHER A
      COMMAND NOR A STATEMENT KEYWORD.  '='  '(' OR
      ',' EXPECTED FOR ASSIGNMENT STATEMENT.
```

Every time you enter a line, BASIC checks it for the **syntax**[4] it expects, and if it doesn't find what it expects, sends you a message like the one above. Although you may not understand the message yet, the fact that you get it means you made an error.[5]

BASIC puts a **flag** (the 1) under the opening quote, and tells you it doesn't understand what follows the 1. In this instance, you made the error in spelling PRINT, and BASIC recognizes that something is wrong. A **keyword** is a name that has a special meaning to BASIC, and it assumes, because PZINT is not a keyword, it must be the name of a variable in an assignment statement. We'll discuss variables and assignment statements later. Forget about the rest of the message; an explanation of it probably wouldn't help you much at this stage of your BASIC programming education. However, messages like this one will help you to **debug**, or find the errors in, the programs you write later.

Sometimes you can make more than one error on a line, and BASIC will flag each one with a separate number. The message you receive is keyed to that number, as in the previous example:

```
###1) BAS30013S...
```

where the ###1 is keyed to the "1" under the quote.

One way to correct the error is merely to retype the line correctly:

```
 * print 'hello'
hello
```

We'll tell you about other ways in "Calculating Unit Prices" on page 10.

---

4 Syntax is really the *grammar* of the BASIC language. The syntax check that BASIC does at this point reveals certain kinds of errors, but might not find others. That is, you could type in a statement that was syntactically correct but that didn't make sense. You could compare it to an English sentence like "The car waltzes beautifully," that is grammatically correct, but doesn't make sense (not in real life, anyway). When you *run* your program, BASIC will make a "sense" check on the statements you've typed in.

5 BASIC does provide some help in interpreting error messages. If, when you have an error message on your screen, you enter the command HELP on the entry line, BASIC will clear your screen and replace it with a **help screen** that will explain the error message to you. To get back to BASIC from the help screen, press the PF3 or PF12 key.

BASIC provides other kinds of help, as well. We'll describe them in more detail later.

you enter ▶

```
* print 5
```

Here's what happens:

```
* print 5
5
*
 _
```

```
 * print 'hello'
hello
 * pzint 'hello'
        1
###1) BAS30013S PRECEDING IDENTIFIER IS NEITHER A
      COMMAND NOR A STATEMENT KEYWORD.  '='  '(' OR
      ',' EXPECTED FOR ASSIGNMENT STATEMENT.
 * print 'hello'
hello
 * print 5
 5
 * print 5+2
 7
 * print 2-5
-3
 * print 5*2
 10
 * print 5/2
 2.5
 *
  _
```

Figure 1. BASIC Screen. Here's what your screen should look like so far.

Doing Immediate Calculations — Using BASIC Like a Calculator

Now let's ask BASIC to print a number. Enter:

```
* print 5
```

Because 5 is a number, not a message like HELLO, it is not enclosed in quotes. In programming terms, this is called a **numeric constant.**

Now, again, BASIC does what it's told:

```
5
*
 _
```

You can tell BASIC to do simple arithmetic and to print the answer, just as you might use a calculator. Entering the following lines gives you the results shown:

```
* print 5+2
7
* print 2-5
-3
*
 _
```

Use an * to indicate multiplication and a / for division:

```
* print 5*2
10
* print 5/2
2.5
*
 _
```

If you have completed all the above steps, and have entered everything at the terminal, the lines on your screen will look like those shown in Figure 1. Depending on the size of your screen, some of the lines might have scrolled off the top.

Now that you've tried immediate statements, it's time to do something a little more complicated — write a program. Actually, you could make a program out of the immediate statements you've just typed in. If you had a lot of calculations to do, the advantage of a program would be that the computer would do the calculations together, instead of one at a time. Such a program would look like this:

```
100 print 5+2
110 print 2-5
120 print 5*2
130 print 5/2
140 end
```

The numbers (100, 110, 120, and so forth) in front of the statements are called **line numbers.** We'll explain them in the next chapter.

BASIC would print the answers like this:

```
 7
-3
 10
 2.5
```

However, that's not the kind of program that people usually write in BASIC. We'll explain more about programs in Chapter 2.

Summary

In this chapter, we told you how to get into BASIC and how to leave it. We showed you how to give BASIC some problems to do that will produce results immediately, as if you were using BASIC like a pocket calculator. We told you what to expect BASIC to do if you make a syntax error in typing in one of the immediate statements. We also presented the following commands and statements:

Command	What It Does
BASIC	Allows you to connect with IBM BASIC
HELP	Displays information about BASIC
QUIT	Ends your current IBM BASIC session

Statement	What It Does
PRINT	Displays data on your screen

Chapter 12, "Command and Statement Summary" on page 79 contains a complete summary of all statements and commands used in this book.

What To Do Next

Quit out of BASIC and log off your computer

OR

go on to Chapter 2.

Chapter 2. Developing a Program

The KETCHUP Algorithm

1. Divide the price of bottle 1 by the number of ounces in bottle 1 (39/12).
2. Divide the price of bottle 2 by the number of ounces in bottle 2 (55/15).
3. Divide the price of bottle 3 by the number of ounces in bottle 3 (47/14.5).
4. Look at the three answers and select the lowest one.

In this chapter you'll learn what a program is and how to write a simple one. You'll type in your program, correct errors, save the program in a permanent file, and run it to get results. Then you'll delete some of the lines, and save it and run it again.

What a Program Is

Here is a problem. A supermarket is selling bottles of ketchup in three sizes. One is 12 ounces and is selling for 39¢,[6] another is 15 ounces and is selling for 55¢, the last is 14.5 ounces and is selling for 47¢.[7]
How do you know which is the best buy?

You know the way to solve this problem is to find out the price per ounce for each of the three bottles, and compare the results. Instead of going ahead and doing this arithmetic, we'll make an orderly list of just what the process of comparing unit prices involves.

1. Divide the price of bottle 1 by the number of ounces in bottle 1 (39/12).
2. Divide the price of bottle 2 by the number of ounces in bottle 2 (55/15).
3. Divide the price of bottle 3 by the number of ounces in bottle 3 (47/14.5).
4. Look at the three answers and select the lowest one.

What we've done is to write an **algorithm,** in English, for finding the cheapest bottle of ketchup. *Any set of instructions to solve a particular problem is called an algorithm.* Once you have the algorithm for a problem written down, it's easy to translate it to a **program**, which is something that BASIC can understand.

This is exactly what you're going to learn to do in this book. You're going to write programs, or lists of instructions, but they are going to be in BASIC instead of English.

Look back at the algorithm for finding the unit price of ketchup. While this list tells you how to go about finding the unit price, it doesn't actually give you an answer until *you* perform the steps. This is true of a BASIC program, too. The program is an orderly plan for solving the problem, but it doesn't give you an answer until the computer *performs* it; that is, until you tell the computer to *run* the program.

---

6 When dealing with numbers that represent money, don't use the dollar sign ($) or cent sign (¢) in the BASIC program.

7 For nostalgic reasons we've kept the 1974 prices from the original *B is for BASIC*.

The KETCHUP Program

```
100 let a=39/12
110 let b=55/15
120 let c=47/14.5
130 print a
140 print b
150 print c
160 end
```

Calculating Unit Prices

The BASIC program looks a lot like the algorithm. One of the differences is the line numbers. BASIC requires that each line in a program be numbered (that's how it knows the order in which the statements are to be processed.)

Now it might seem logical to number the first line 1, the second 2, and so forth. But, as a programmer, you'll find that it is occasionally necessary to insert statements into an already-written program. And, because BASIC allows only whole-number line numbers, how do you insert something between 3 and 4? Or something before 1? You can't. So you can see that it would *not* be a good idea to number the lines 1, 2, and 3. Most programmers begin their programs with either 10 or 100 and use an increment of 10 for each successive line number. Let's start with 100.

Here's the program:

```
100 let a=39/12
110 let b=55/15
120 let c=47/14.5
130 print a
140 print b
150 print c
160 end
```

Now we'll see why it looks that way.

The first three lines in the BASIC program are:

```
100 let a=39/12
110 let b=55/15
120 let c=47/14.5
```

These three lines do the arithmetic of the program. There is a line number for each line, followed by a LET statement. The LET statements define the arithmetic you want done.

A, B, and C are called **variables**. Variables are identifiers for items that can change in value each time you use them. They wouldn't be absolutely necessary in this program (for example, you could use a PRINT statement — `100 PRINT 39/12` — to get the unit price for the first bottle of ketchup). However, you'll see the advantage of using variables as your programs get more complex.

If you were to tell the computer to go ahead and execute lines 100, 110, and 120, it would, but there's a catch. You haven't told it to tell you what the answer is. The computer only does what you tell it to do, so you have to write program instructions to tell it to give you the values of A, B, and C:

```
130 print a
140 print b
150 print c
```

Again, there is a line number and then a PRINT statement. PRINT tells the computer to type out values. Now the program will find the values of A, B, and C according to the instructions in lines 100, 110, and 120, and will print the values at your terminal, just as you used PRINT in Chapter 1 to print the number 5 and the value of 5/2.

KETCHUP is a simple program. It doesn't actually select the best buy. However, you can see for yourself which unit price is the lowest.

Since this is the end of program KETCHUP, you add a final statement:

```
160 end
```

You put an END statement at the end of all your programs. It lets BASIC know a program is complete.

You have an entire program written in BASIC. Now we'll tell you how to type it in line by line, but first, let's get rid of all the junk you've accumulated in your workspace.

What Is A Workspace?

Whenever you're writing programs in BASIC, you are assigned a **workspace** for your exclusive use. A workspace is an area within the computer that contains the BASIC program statements you are currently working on. You can fill the workspace, save its contents, erase or change the contents, and retrieve statements you have previously saved.

Whatever programming work you do with BASIC, you do through your workspace. Think of it as a blackboard. Like things written on a blackboard, the programs you create in your workspace are temporary. That is, unless you specifically prevent it, they are automatically erased when you end the BASIC session. Each new BASIC session gives you a clean blackboard to work on.

You type program statements into your workspace using the BASIC editor. The services of the editor are available every time you connect with BASIC.

One important thing to remember: your terminal screen is *not* your workspace. Program statements can be displayed on your *terminal screen* and not be in your *workspace*, and vice versa.

Clearing Your Workspace

you enter ▶

```
* print 5/2
2.5
* initialize
```

Here's what happens:

```
* print 5/2
 2.5
 * initialize
INITIALIZATION COMPLETED.
THE WORKSPACE DOES NOT HAVE A NAME.
 *
```

Getting Rid of the Junk — The INITIALIZE Command

(Have you been in the BASIC environment since Chapter 1 — do you still have a screen full of immediate statements? If so, read this section and follow the instructions for clearing your workspace. If you just logged on and connected with BASIC, you have a nice, clean workspace and don't need to follow the instructions in this section, but it won't hurt anything if you do.)

Remember we told you your workspace was like a blackboard? Well, you've already used up a lot of chalk writing immediate statements on it. When you've been trying things out in BASIC, and you want to get down to writing a real program, it's a good idea to erase the blackboard. You do that with the INITIALIZE command, which simply clears your workspace. Type:

```
* initialize
```

and press ENTER.

You should get the following message:

```
INITIALIZATION COMPLETED.
THE WORKSPACE DOES NOT HAVE A NAME.
```

Your workspace is now clear and ready for you to type in the KETCHUP program. INITIALIZE doesn't clear your screen, so some immediate statements and their results probably remain on the screen. Don't worry about them; they'll scroll off the top of the screen eventually. If you really want a clear screen, type PRINT NEWPAGE on the entry line.

Typing In a Program — First Statement

```
 * print 5/2
  2.5
 * initialize
INITIALIZATION COMPLETED.
THE WORKSPACE DOES NOT HAVE A NAME.
 * 100 let a=39/12
```

you enter ▶

Here's what happens:

```
 * print 5/2
  2.5
 * initialize
INITIALIZATION COMPLETED.
THE WORKSPACE DOES NOT HAVE A NAME.
 * 100 let a=39/12
 *
  _
```

Creating a Program

You're going to create your program using the BASIC editor. You've already used the editor in Chapter 1 to type in immediate statements. Now you're going to use it to create a program. BASIC will know that the statements you're about to type in are program statements and not immediate statements, because you're going to start each statement with a line number. BASIC *program statements must always have line numbers in front of them; the line numbers become part of the statements.*

To enter the first statement in your program, type a 100, a blank space, and then the rest of the line:

```
 * 100 let a=39/12
```

Then press ENTER.

We'll assume you typed the line correctly. If you did, your screen should look like this:

```
 * 100 let a=39/12
 *
  _
```

Note: Extra blanks in a BASIC statement are okay, so you could also type the line like this:

```
 * 100 let a = 39/12
 *
  _
```

or like this:

```
 * 100 let a = 39 / 12
 *
  _
```

BASIC is now ready for you to type in the next line of the program, which you're going to number 110. Enter it the same way you did line 100. Don't forget to include the line number!

```
 * 110 let b=55/15
```

Now type in the other lines, one by one:

```
120 let c=47/14.5
130 print a
140 print b
150 print c
160 end
```

you enter ▶

```
* list
```

Here's what happens:

```
 * list
100 let a=39/12
110 let b=55/15
120 let c=45/14.5
130 print a
140 print b
150 print c
160 end
 *
  -
```

Displaying a Program — The LIST Command

Let's make sure BASIC has the whole program in your workspace just the way you want it (remember, what's on your *screen* isn't necessarily what's in your *workspace*). To get a **listing** of the program as BASIC has recorded it, type:

```
* list
```

and press ENTER.

BASIC will display on your screen all the lines in the program. Assuming you've finished typing statements, that will be the entire program.

```
100 let a=39/12
110 let b=55/15
120 let c=47/14.5
130 print a
140 print b
150 print c
160 end
```

You can also list a range of lines. To list lines 100 through 120, type:

```
* list 100 to 120
```

```
* print 5/2
2.5
* initialize
INITIALIZATION COMPLETED.
THE WORKSPACE DOES NOT HAVE A NAME.
* 100 let a=39/12
* 110 let b=55/15
* 120 let c=45/14.5
* 130 pzint a
```

you enter ▶

Here's what happens and what to do about it:

```
* 100 let a=39/12
* 110 let b=55/15
* 120 let c=45/14.5
* 130 pzint a
            1
###1) BAS30013S PRECEDING IDENTIFIER IS NOT A
      STATEMENT KEYWORD. '=' '(' OR ','
      EXPECTED FOR ASSIGNMENT STATEMENT.
* 130 print a
```

OR:

```
* 100 let a=39/12
* 110 let b=55/15
* 120 let c=47/14.5
* 130 pzint a
            1
###1) BAS3001 etc.
*
* -
```

The cursor is here.
Using the cursor-control keys on your keyboard, move it up here, and type an r in place of the z. Then press ENTER. BASIC *will substitute the corrected line in your workspace, will put the cursor back on the entry line, and will put the message* 1 LINE(S) EDITED. *in the bottom right-hand corner*:

```
                                    1 LINE(S) EDITED.
```

Correcting Errors

We hope you now have the complete program typed in, and didn't make any mistakes while you were doing it. But suppose you make an error in line 130 like the one described in Chapter 1:

```
 * 130 pzint a
            1
###1) BAS30013S PRECEDING IDENTIFIER IS NOT A
      STATEMENT KEYWORD.  '='  '(' OR ','
      EXPECTED FOR ASSIGNMENT STATEMENT.
```

As we showed you in Chapter 1, you can correct the error by retyping the line:

```
 * 130 print a
```

You also have another choice: simply move the cursor up to where BASIC has displayed line 130, and type the correct letter (r) over the incorrect one (z). See the last figure on the opposite side of the page for an illustration of the process.

Note: There are some restrictions with BASIC **full-screen editing**. For example, you can't delete or add lines this way, and you can't edit immediate statements or commands. BASIC's way of telling you about this kind of restriction is to prevent you from typing by locking the keyboard. If that happens, press the **reset key**, move the cursor, and try doing what you want to do another way. It shouldn't take you much time to get used to the BASIC editor.

Beginning BASIC users often make the mistake of moving the cursor with the cursor-control keys to the position next to the asterisk prompt on the entry line, and then trying to type. BASIC won't let you type in that space. The easiest way to get BASIC to move your cursor back to the proper position on the entry line is to press the ENTER key. The cursor will then jump to the correct position, one space to the right of the asterisk.

Saving a Program

you enter ▶

```
* save ketchup
```

Here's what happens in VM:

```
* save ketchup
PROGRAM SAVED IN FILE 'KETCHUP BASIC A1'.
* _
```

Here's what happens in MVS:

```
* save ketchup
PROGRAM SAVED IN FILE 'userid.KETCHUP.BASIC'.
* _
```

Note: userid is the user identifier you use when you're logging on to the system.

Putting a Program in a Permanent File — The SAVE Command

You're going to run KETCHUP (and find out if it works), but first it would be a good idea to save a copy of the program in a permanent file (remember that when you leave BASIC, your workspace and its contents will disappear). Think of a file as a file folder and its contents that get tucked away in a drawer. To save your KETCHUP program, type:

```
* save ketchup
```

and press ENTER.

Note: You can name your programs anything you want to, but names can be no longer than 8 characters, and can't contain blanks.

BASIC responds with:

```
PROGRAM SAVED IN FILE 'KETCHUP BASIC A1'.
 * _
```

if you're using VM, and:

```
PROGRAM SAVED IN FILE 'userid.KETCHUP.BASIC'.
 * _
```

if you're using MVS.

The response indicates that your file was saved, and BASIC is waiting for further instructions.

At this point there are *two* copies of your KETCHUP program, one in your workspace, and the other in a permanent file.

To be sure the file is there, enter the command:

```
query ketchup
```

Information about the KETCHUP file should be listed on your screen. The command `QUERY PROG` will produce a list of all your BASIC programs.[8]

---

8 In the BASIC environment, you can get rid of the permanent KETCHUP file by issuing the command:

```
purge ketchup basic
```

Outside of BASIC, you can display or erase BASIC files just the way you do other CMS or TSO files.

Running a Program

you enter ▶

```
* run
```

Here's what happens:

```
* run
3.25
3.66667
3.24138
END AT LINE 160.
*
  -
```

Running a Program — The RUN Command

Now that you've saved the program in a permanent file, tell the computer to run KETCHUP. Type:

```
* run
```

and press ENTER. You'll get this response:

```
 3.25
 3.66667
 3.24138
END AT LINE 160.
 *
  -
```

Here's what happened: BASIC had the computer run KETCHUP. The values of A (3.25), B (3.66667), and C (3.24138) were printed when lines 130, 140, and 150 were executed, and now that the program has ended, BASIC is waiting for you to tell it what to do next.[9]

9 Remember we told you in Chapter 1 that when your program is run BASIC does a "sense" check on the program statements. Therefore, BASIC might discover new errors that didn't appear when it did the syntax check on each statement earlier. When it discovers errors at "run-time," it often gives you a choice: RUN WITH ERRORS: YES/NO?. It's up to you: you can answer NO, and go back and try to figure out what you did wrong, or let the program run and try to discover from what happens what you need to change. You also get this error message if you don't correct errors you've made that BASIC told you about when you typed in the statement. These are syntax errors.

Deleting a Line from a Program

you enter ▶

```
* delete 120
```

Here's what happens:

```
* delete 120
LINE 120 DELETED.
*
```

Deleting Program Statements — The DELETE Command

Suppose the next time you go to the store the third brand of ketchup is no longer on the shelf. You'll need to delete part of your program to make it reflect reality. First, list the program to see what lines you want to delete.

```
* list
100 let a=39/12
110 let b=55/15
120 let c=47/14.5
130 print a
140 print b
150 print c
160 end
```

You have no need to calculate the unit price of C, or to print it, so lines 120 and 150 can go. Enter the following statement:

```
* delete 120
```

BASIC will respond:

```
LINE 120 DELETED.
*
```

Now delete line 150 the same way:[10]

```
* delete 150
LINE 150 DELETED.
*
```

You can also delete a line number by merely typing the line number, with no text following it, and pressing ENTER.[11]

---

10 The space between the DELETE command and the line number is necessary. You *can't* say DELETE150.

11 *Be careful!* Until you get used to this way of deleting lines, it's easy to lose a line you really intended to edit and not delete.

Renumbering Your Program

you enter ▶

```
* renumber
```

Here's what happens:

```
* renumber
RENUMBER COMPLETED.
* _
```

Renumbering Your Program — The RENUMBER Command

You've corrected your program, but your line numbers aren't neatly numbered any more in increments of 10. To verify that it's true, list the program again:

```
* list
100 let a=39/12
110 let b=55/15
130 print a
140 print b
160 end
```

BASIC keeps the original line numbers of program statements unless you tell it otherwise. Even though programs will still run correctly with sequence numbers missing, programmers often renumber their programs after they've been working on them awhile, because it makes them neater and easier to read. To do that, enter:

```
* renumber
RENUMBER COMPLETED.
```

Now if you list your program it should look like this:

```
100 let a=39/12
110 let b=55/15
120 print a
130 print b
140 end
```

Resaving a Changed Program

you enter ▶

```
* save ketchup
```

Here's what happens in VM:

```
* save ketchup
PROGRAM SAVED IN FILE 'KETCHUP BASIC A1'.
*
```

Here's what happens in MVS:

```
* save ketchup
PROGRAM SAVED IN FILE 'userid.KETCHUP.BASIC'.
*
```

Resaving a Changed Program — The SAVE Command

Unless you expect the third brand to reappear on the grocery shelves, you'll probably be satisfied with keeping your KETCHUP program the way it is now. Remember that the copy in the permanent file is the *old* program; only the copy in your workspace is the *new* program. Therefore, you probably want to resave the program, so that the copy in the permanent file is up to date.

To do that, you again direct BASIC to save the KETCHUP program by typing:[12]

```
* save ketchup
PROGRAM SAVED IN FILE 'KETCHUP BASIC A1'.
```

or

```
PROGRAM SAVED IN FILE 'userid.KETCHUP.BASIC'.
```

12 Under certain circumstances, you might also get a prompt like this: `REPLACE EXISTING FILE (YES/NO)?`. If you're sure the program you're saving should replace another file by the same name (if, for example, you've updated an existing program), you should answer `yes` to the prompt. Otherwise, you should answer `no` to the prompt, and save your program using a different name.

Summary

In Chapter 2:

1. You typed the KETCHUP program into your workspace line by line.
2. BASIC checked each line for syntax errors.
3. If you made errors, you corrected them before going on.
4. You listed the program to make sure it was correct.
5. You saved the program in a permanent file.
6. You had the computer run the program, and you checked the results.
7. You deleted two lines from your program, renumbered it, and then resaved it.

You learned these commands and statements in Chapter 2:

Command	What It Does
DELETE	Removes statement lines from your workspace
INITIALIZE	Clears your workspace
LIST	Displays program lines on the screen
PURGE	Removes files from your library
QUERY	Permits interrogation of your user files
RENUMBER	Renumbers statement lines in your program
RUN	Directs BASIC to run your program
SAVE	Puts a copy of your program into a permanent file

Statement	What It Does
LET	Assigns values to variables

Chapter 12, "Command and Statement Summary" on page 79 contains a complete summary of all statements and commands used in this book.

Exercise — CHECKBAL Program

Write a program to solve the following problem. Follow these steps (refer back to the pages indicated, if you've forgotten how):

1. Look at the problem and algorithm below to figure out how to write the program.
2. Clear your workspace (as explained on page 12).
3. Type in the program (as explained on page 13).
4. Save the program under the name CHECKBAL (as explained on page 16).
5. Run the program to get the results (as explained on page 17).

The algorithm appears below, and the program is in Appendix A, "Answers to Exercises" on page 81.

Problem: Find out how much your checking balance is this month. You had a $97.50 balance in your checkbook last month. You made these deposits:

```
$15.00  $68.75
```

You wrote these checks:

```
$10.75  $54.00  $37.50
```

There was a 10¢ service charge for each check you wrote.

Algorithm:

1. Add the deposits to the balance. Call that amount A.
2. Add the checks. Call that amount B.
3. Subtract the total of the checks and the 30¢ service charge (10¢ for each of three checks) from the total in 1. Call that amount C.
4. Print the final balance.

Note: Remember, when you're dealing with numbers that represent money, don't use $ or ¢ in your program.

What To Do Next

Quit out of BASIC and log off your computer.

Chapter 3. More Programming and Editing Techniques

In this chapter, you're going to learn how to get an old program out of the permanent file to work on. You'll list the program, add some new lines to it, renumber it, and resave it. Then you'll document your program with remarks. We'll ask you to type in a new program, a process we'll make easier by having BASIC number the lines for you.

Before you begin this chapter, log on to the computer and connect with BASIC. If you've forgotten how to do that, look back at page 2.

Loading in an Old Program —The LOAD Command

Suppose last week a new brand of ketchup appeared on your grocer's shelves. You'd like to see how the unit price of the new brand compares with the unit prices you calculated with your KETCHUP program. To do that, you'll need to modify the KETCHUP program.

As we told you in Chapter 2, when you first connect with BASIC you have an empty workspace. Any programs you've already created are in permanent files, but not yet available to you to edit. If your reason for using BASIC is to make additions or corrections to a previously-written program, you'll first have to load it into your workspace. To bring the KETCHUP program into your workspace, enter:

```
* load ketchup
```

BASIC should respond:

```
5 LINE(S) LOADED. 'KETCHUP' IS THE WORKSPACE NAME.
```

What Might Go Wrong?: (Skip this section if everything worked as described above.)

If you got the message above, but the number of lines loaded is a number other than 5, you didn't follow the instructions for deleting, renumbering, and resaving lines in Chapter 2 correctly. Your program should look like the one listed on the bottom of page 19.

If, while you were working on the KETCHUP program in Chapter 2, you forgot to save the program, or if the program didn't actually get saved, you'll get the following error message:

```
##### BAS00128S PROGRAM FILE 'KETCHUP' ERROR
(FILE CANNOT BE FOUND).
COMMAND IGNORED.
```

To correct the problem, type in the program as it appears on the bottom of page 19. Then save it with the command:

```
* save ketchup
```

Back to the KETCHUP Program: We'll assume you now have the KETCHUP program in your workspace. Here's how it should look:

```
* list
100 let a=39/12
110 let b=55/15
120 print a
130 print b
140 end
```

Adding Program Statements: You'll call the variable for the new brand C. The price is 49 cents for 14 ounces. You'll need a new line after 110 that looks like this:

```
let c=49/14
```

What line number should you use? Any number between 110 and 120 would be okay. Since you know you'll only be adding one line at that place in the program, let's use 115. (If you knew you'd be adding several, you might start with 111.)

Enter line 115:

```
* 115 let c=49/14
```

You also need a new line between 130 and 140. Call it 135 and enter it:

```
* 135 print c
```

Now, as you did with the changed program in Chapter 2, renumber it, list it again, and run it. You probably remember how to do that. If not, look again at information in Chapter 2, pages 19 and 17.

Putting Comments in Your Programs — The REM Statement

You can make your programs easier to work with, and easier for other people to use, if you include, right in the programs, descriptions of what the programs do. The descriptions are known as **remarks**. You write remarks as if they were statements or pieces of statements in the BASIC program, but they don't serve any function as far as the execution of the program goes: They are there solely for information. You can place them at the beginning, or anywhere within a program.

There are two kinds of remarks:

1. Remarks that take up an entire program statement line

   These remarks begin with a statement line number (as do all program statements) followed by "rem", followed by any text you choose. Here's an example:

   ```
   100 rem This program computes batting averages
   ```

2. Remarks that *follow*, and are on the same line as, a normal program statement.

   These remarks usually comment only a single line. They follow the program statement, and are separated from it by an exclamation point:

   ```
   130 let c=a-b-.30 ! Balance minus withdrawals minus service charge
   ```

It might be a good idea to put comments in the KETCHUP program, so if you look at it six months from now, you'll remember what it was all about. Put in the program name and a general remark at the top. Since you started the program with line 100, call the new lines 90 and 95:

```
90 rem KETCHUP program
95 rem Calculates unit prices for 3 brands of ketchup
```

As you remember from modifying the KETCHUP program in Chapter 2, you have a choice in changing existing lines: Type the line over again, or list the program, move the cursor to the place where you want to type, and make the change. See page 15 if you've forgotten how.[13]

Put in whatever other comments you think are meaningful. Here's a suggestion:

---

13 When you move the cursor up to type in the ! comments, you must first position the cursor at the space immediately following the 12 (in line 100) and the a (in line 130), press the space bar to indicate that a blank should appear there, and then type the !. If you move the cursor directly to the spot where you want the ! to appear and start typing, the ! and following text will shift one space left when you press ENTER.

```
90  rem KETCHUP program
95  rem Calculates unit prices for 3 brands of ketchup
100 let a=39/12 ! Calculates the unit price of brand a
110 let b=55/15
120 let c=47/14.5
130 print a ! Prints the unit price of brand a
140 print b
150 print c
160 end
```

Now renumber your program, resave it, and run it once more to make sure you didn't change something you didn't mean to.

Automatic Line Numbering — The AUTO Command

We're going to give you a new program to type in (**don't forget to initialize before you do it!**), but before we do that we'd like to show you a BASIC feature that saves your typing keystrokes and ensures that every line has a correct line number. That feature is called **automatic line numbering**. When you're using it, BASIC prints line numbers for you, starting with 100, and it increments by 10 as you get to new lines, just as you probably would do yourself.

(Read this section, but don't type anything new until you get to the DENTBILL program below.)

You get into **AUTO mode** by giving the AUTO command:

```
* AUTO
```

to which BASIC responds:

```
* 100 _
```

You then merely type in the program line as you have been doing. A new line number prompt will appear every time you press ENTER.

What Happens After an Error in AUTO Mode

If you make a syntax error while in AUTO mode, you get an error message, and AUTO mode stops. For example:

```
* 100 let a==39/12
              1
###1) BAS30122S NUMERIC TERM EXPECTED.
* _
```

Because you are no longer in AUTO mode, you can correct your error by retyping the statement, including the line number:

```
* 100 let a=39/12
```

or by moving the cursor up to the extra equal sign and deleting it.

To reenter AUTO mode, you type AUTO again:

```
* AUTO
```

and BASIC responds:

```
* 110 _
```

Notice the line number is 10 greater than your last statement.

Getting Out of AUTO Mode

When you want to end AUTO mode, you simply press ENTER without typing anything after the AUTO number on the line. For example:

```
* 170 (press ENTER)
* _
```

and, as you can see, you have exited from AUTO mode. (Careful; don't accidentally type any character or BASIC will think you want *that* to be line 170. It will return with an error message, exit AUTO mode, and expect a command.)

Line 170 won't be part of your program. List the program to check, if you like.

Calculating Insurance Coverage for a Dental Bill — The DENTBILL Program

Here's the new problem:

Problem: Last month you went to the dentist and had an examination and x-rays. That cost $25. You had two teeth filled. That cost $24. Your insurance will pay for 75% of everything over $15. How much do you have to pay, and how much does the insurance pay?

Algorithm

1. Find the total dentist bill (D)
2. Subtract $15.00 to find the amount eligible for insurance (call it E).
3. Take 75% of the result (call it I). That's how much the insurance pays.
4. Subtract the insurance money from the total bill D to find out how much money you owe (call this M).
5. Print out how much you have to pay and how much the insurance will pay (M and I).

The BASIC Program

```
100 rem DENTBILL program
110 rem Calculates insurance coverage for a dental bill
120 let d=25+24  ! D is the total dental bill
130 let e=d-15   ! E is the amount eligible for insurance
140 let i=.75*e  ! I is the amount the insurance pays
150 let m=d-i    ! M is how much money you owe
160 print m,i
170 end
```

Notice the PRINT statement in line 160. Since you want to know both your payment and the insurance payment, you can specify both M and I in the one statement. Any time you want to print more than one value, you can do it with a single PRINT statement if you list the values, separated by commas.

Now type in the program and run it. Follow these steps (refer back to the pages indicated, if you've forgotten how):

1. Clear your workspace (as explained on page 12).
2. Type in the program (as explained on page 13) — use AUTO mode (as explained on page 25).
3. Save the program under the name DENTBILL (as explained on page 16).
4. Run the program to get the results (as explained on page 17). M is 23.5 and I is 25.5.

Summary

In this chapter you learned how to load an old program into your workspace from the permanent file, and how to add comments to your programs.

You learned the following commands:

Command	What It Does
AUTO	Numbers your program statements automatically.
LOAD	Gets a program from a permanent file and puts it into your workspace.

Figure 2. Commands Presented in Chapter 3

You also learned the following statements:

Statement	What It Does
! (exclamation point)	Inserts remarks into your programs (usually follows normal program statements).
REM	Inserts remarks into your programs.

Figure 3. Statements Presented in Chapter 3

Chapter 12, "Command and Statement Summary" on page 79 contains a complete summary of all statements and commands used in this book.

You now know all the fundamentals of programming with BASIC. You know what a program is, how to go about writing one, how to enter one, save it, run it, and get it later when you want it. In the next chapters, we'll concentrate on using BASIC to write more useful programs.

Exercises

Write programs to solve the following problems. Include appropriate remarks. Save the programs and run them. The algorithm for program 1 appears below, and the programs are in Appendix A, "Answers to Exercises" on page 81.

Problem 1 — ALGEBRA Program

Find X in this algebra problem where the value of Y is 7.

```
x=.3y-1.5
```

The Algorithm

1. Solve the equation.
2. Print X.

Note: Remember to use the * symbol for multiplication.

Problem 2 — LAWN Program

Find out how much sod Mr. Harris has to buy to cover his 70x80 foot lawn. If sod is 11.5¢ a square foot, how much will it cost?

Problem 3 — HAMBURG Program

You are supplying hamburgers for a party. There will be 17 adults and 14 children. Each hamburger will have three ounces of meat in it. Kids can eat two hamburgers each, adults three. How much meat do you have to buy?

Problem 4 — CHARTER Program

You are arranging a charter flight. The plane costs $10,000 to hire. There are 117 seats on it. You want to add on a 5% surcharge for profit and a $2.50 per person airport tax. How much is each ticket?

Problem 5 — SOAP Program

Dr. Lemming sees an average of 60 patients a week in his office. Before he sees each patient, he washes his hands with a liquid antiseptic soap that he buys in gallon jugs. Each time he washes his hands, he uses a teaspoon of soap. How many weeks will the gallon last? If he also uses half an ounce of hand cream at the end of each day, seven days a week, to keep his hands from chapping from all the washing, how much hand cream does he use up with each gallon of soap?

Note: There are 6 teaspoons an ounce, 32 ounces a quart, 4 quarts a gallon.

What To Do Next

Quit out of BASIC, or go on to the next chapter.

Chapter 4. BASIC Arithmetic

Many of the programs you'll be writing will involve arithmetic. KETCHUP used three LET statements involving arithmetic:

```
100 let a=39/12
110 let b=55/15
120 let c=47/14.5
```

Obviously, the more freedom you have to write complicated arithmetic equations, the easier it is to write BASIC statements to solve problems. Just about anything you can write in an algebraic expression, you can express in BASIC.

In BASIC, there are five numeric operations, each with its own symbol.

For This Numeric Operation	Type This
addition	+
subtraction	-
multiplication	*
division	/
raising a number to a power (exponentiation)	** (use * twice)

For example:[14]

X+Y is typed x+y
X^2 is typed x**2
3B is typed 3*b
A÷4 is typed A/4

Be careful always to include the * when you mean multiplication. 3B is all right in math, but the computer only understands that you want 3 times B if you type it as 3*b.

Priorities in Numeric Expressions

When you combine several numbers and several numeric operations in a LET statement, the computer doesn't necessarily perform the arithmetic in the order you type it on the line. Instead, built into BASIC are priorities for performing different numeric operations. When there is more than one numeric operation in a LET statement, the computer does the arithmetic in this order:

exponentiation first (**)
multiplication and division second (* and /)
addition and subtraction last (+ and -)

If there's more than one operation of the same priority in a statement, the computer does the arithmetic for those operations from *left to right*.

For example, if you want to compute $3Y^2$, you type it in as 3*y**2, the computer first squares Y, then multiplies the result by 3, since exponentiation has a higher priority than multiplication. This means there is never any question about whether you mean $3Y^2$ or $(3Y)^2$. As far as the computer is concerned, you always mean $3Y^2$ if you write 3*y**2.

---

14 Remember you can use lowercase, uppercase, or mixed case letters in BASIC.

Using Parentheses to Change Priorities and for Clarity

Suppose you *do* mean $(3Y)^2$, however. You can change the automatic priorities by using parentheses. Parentheses work the same in BASIC as they do in math. They group items together so that the computer knows what numeric operation you want done to what numbers. So to get $(3Y)^2$, you *must* type (3*Y)**2 to avoid getting $3Y^2$.

You can always use parentheses *instead* of the built-in priorities. So even though you *can* type $3Y^2$ as 3*Y**2, you can, if you prefer, type it in as 3*(Y**2).

We recommend that you use parentheses for clarity even if you aren't trying to change priorities. It makes your arithmetic statements more explicit and easier to read.

Some Examples of Numeric Expressions

Here are some examples of numeric expressions.

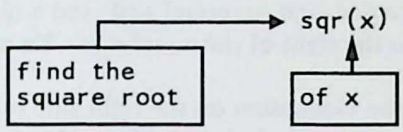
This is the way it looks as arithmetic	This is the way it looks in BASIC	This is what it means
$\frac{a+b+c}{2}$	$(a+b+c)/2$	First add a, b and c; divide the whole sum by 2.
$a+\frac{b+c}{2}$	$a+((b+c)/2)$ *or* $a+(b+c)/2$	Add b and c; divide the sum by 2; add the result to a.
3a+4	3*a+4	Multiply a by 3; then add 4.
3(a+4)	3*(a+4)	Add a and 4; multiply the whole sum by 3.
x^2+7	x**2+7	Square x and add 7.
$(x+7)^2$	(x+7)**2	Add x and 7; square the whole quantity.
$\frac{(x+1)^2}{2}$	((x+1)**2)/2 *or* (x+1)**2/2	Add x and 1; square the whole quantity; divide the whole result by 2.
$\left(\frac{x^2}{2}\right)\left(\frac{x+y}{3}\right)$	((x**2)/2)*((x+y)/3) *or* x**2/2*(x+y)/3	Square x and divide by 2; add x and y and divide the sum by 3; multiply the two results by each other.

As you can see, the more complicated the arithmetic expression looks, the more complicated the BASIC expression looks. When you're writing BASIC expressions, remember this: *Parentheses must always be in balanced pairs — as many right parentheses as left parentheses.* If a statement looks like it's getting too complicated to handle, you can always break it down into several simpler statements. For example, for the last item in the figure above you could write:

```
a=x**2/2
b=(x+y)/3
c=a*b
```

Square Roots

You can take square roots automatically with BASIC. In BASIC, instead of a symbol $\sqrt{}$, you use the letters SQR followed by a set of parentheses. The item whose square root you want to know goes inside the parentheses:



You can use SQR in any of your BASIC statements, and the things inside the parentheses can involve any kind of arithmetic. For example

$\sqrt{X+Y}$	is written in BASIC	SQR(X+Y)
$\sqrt{\dfrac{X+Y+Z}{5}}$	is written in BASIC	SQR((X+Y+Z)/5)
$A+\dfrac{\sqrt{X}}{2}$	is written in BASIC	A+SQR(X)/2

Here's a simple program using SQR:

```
100 rem SQR program
110 rem Finds the square root of 25
120 x = 25
130 print sqr(x)
140 end
```

SQR is called a **function** or **intrinsic function** in BASIC. There are others; some are listed in Chapter 9, "Built-In Functions" on page 63.

Using Generalized Computations

Many of your BASIC programs will involve arithmetic done with LET statements. At some point, you'll have to give the computer actual numbers with which to solve the arithmetic. But it's not always a good idea to use actual numbers in a LET statement. If you use a *general* LET statement in a program, you can use the same program for different sets of numbers.

In KETCHUP, this means that instead of writing a statement `let a=39/12`, you could include two simple BASIC statements to supply the numbers 39 for the price, `p`, and 12 for the ounces, `n`:

```
100 let p=39
110 let n=12
```

and then add a statement to do the calculation:

```
120 let k=p/n
```

These statements would tell the computer that `p` has the value 39 and `n` the value 12, when it executes the general LET statement in line 120.

You could change the `let b=55/15` statement in a similar manner. Don't use actual numbers in computations — use variables, instead. Just be sure the variables have values that you've set before — perhaps in a LET statement. That way, if you ever want to change the values, you change them in only one place, instead of having to analyze the computation. Be as general as possible. You'll find that you will use the programs you have written over and over again with small changes or with different values for the variables.

How LET Statements Work

When you write a LET statement, it probably looks very much like a mathematical equation to you. But in LET statements, you are talking BASIC, not math, and to BASIC, LET means "assign." A LET statement is made up of three things: a symbol to the left of an equal sign, an equal sign, and a quantity or a computation (called an **expression**) to the right of the equal sign. To BASIC, LET means this:

> Evaluate the expression on the right side of the equal sign first. Then *assign* that value to the symbol on the left side of the equal sign.

In BASIC, you can have a statement like:

```
120 let x=x+1
```

while you couldn't in math, because to BASIC, this statement means to take whatever value X now has, add one to it, and replace the old value of X with this new value.

Incidentally, after you are more familiar with BASIC, you can leave the word LET out of a LET statement without doing any harm. These two statements:

```
100 let x=a+b
100 x=a+b
```

mean exactly the same thing. In all our examples, we'll show the word LET, but it's not necessary to put it in.

A Note about Numbers

When you use actual numbers in BASIC, they can be:

- *Integers* (or whole numbers) like:

  2 -76 842 1000000 999111

- *Decimal numbers* like:

  -1.5 3.7772 .00081 -457.25

- Numbers in *exponential format* like:

  6E7 (meaning $6x10^7$) 5.4E-3 (meaning $5.4x10^{-3}$)

A Note about Variable Names

The identifier of a variable is called a **variable name.** A variable name can consist of up to 40 characters, starting with a letter. Valid characters are the letters of the alphabet, the numbers 0 through 9 and the __ (underscore).

Numeric Variable Names: Numeric variable names are used to represent number variables. For example, HOMERUNS might stand for the number of home runs in a baseball game. SMITH1 might stand for the number of sales Harvey Smith made in January. Figure 4 on page 33 shows examples of valid and invalid numeric variable names.[15]

---

15 Numeric variable names can end with # or % suffix. However, these symbols have special meanings. We don't recommend your using them until you learn more about programming in BASIC.

Valid Names	Invalid Names
X	2
Y%	%Y
X2	X*2
ABC	A B C
AB#	A*B#
VALUE2	VALUE2.2
LOW_BAL	LOW BAL
SMITH1	
HOMERUNS	

Figure 4. Numeric Variable Names — Valid and Invalid

Character Variable Names: While you usually think of variables as standing for numbers, in BASIC you can let a variable stand for combinations of characters, like words or names.

Suppose you have a program that keeps track of sales figures for some sales people. Along with the sales figures, you will probably want to keep a record of the sales people's names. Then you can print out the names with the figures when you run the program. If a variable is going to represent a word or a name, it's called a **character variable.** Character variable names must have a $ suffix. The dollar sign lets BASIC know the variable is a character variable. (It has nothing to do with dollars or money.) The name must begin with a letter. For example, SMITH$ might stand for a salesman named Harvey Smith.

Figure 5 shows examples of valid and invalid character variable names.

Valid Names	Invalid Names
X$	$X
SMITH3$	SMITH3
ABC$	A B C
JOHN_DOE$	JOHN_DOE
SMITH$	

Figure 5. Character Variable Names — Valid and Invalid

In the salesman example,

SMITH$ can be the name of one of the sales people (character data)

SMITH1 can be his sales for January (numeric data)

SMITH2 can be his sales for February (numeric data)

and so on.

To tell BASIC what values are going to be assigned to each of these variables, you write a LET statement. In the LET statement for SMITH$, you write the sales person's name, putting it in quotes. If his name is Harvey Smith, your LET looks like this:

```
120 let SMITH$='Harvey Smith'
```

Character strings are normally 18 or fewer characters.

Summary

In this chapter, you learned how to write arithmetic expressions using BASIC, and how BASIC prioritizes when there is more than one arithmetic operation in a LET statement. We gave you an example (SQR) of a BASIC intrinsic function. We showed you how to make your LET statements serve more general functions. Finally, we explained numeric and character variable names, and showed examples of valid and invalid names.

Exercises

Write BASIC programs to solve one or more of the following problems. Name them as indicated, save them, and run them. The programs are in Appendix A, "Answers to Exercises" on page 81.

Problem 1 — BATTING1 Program

Last season, Slugs Hoolihan, the baseball player, was up at bat 521 times. He made 130 base hits: 15 home runs, 6 triples, 21 doubles, and the rest singles. What is Slug's batting average? What about his slugging average?

Note: The batting average is the number of hits divided by the number of times at bat; the slugging average is:

```
(home runs*4)+(triples*3)+(doubles*2)+(singles*1)
        number of times at bat
```

Problem 2 — ELECTION Program

Here are the results of an election:

Candidate	District 1	District 2	District 3
Adams	4106	2890	2981
Biller	3729	3515	2493
Campbell	4002	3131	3012

Figure out the total vote and what percent of the total each candidate received.

Problem 3 — AIRCOND Program

You are shopping for an air conditioner. You need one between 5000 and 6000 BTUs. You want to find the one with the highest energy efficiency ratio (EER) because it will be the most economical to operate. The EER is defined as:

```
number of BTUs
number of watts
```

A	5000 BTUs	820 watts
B	6000 BTUs	910 watts
C	5500 BTUs	850 watts

Which model should you buy?

Problem 4 — RENTAL Program

You are renting a car to make a one-day trip of about 275 miles. Here are the rates for three companies:

Acme Rent-a-Car	$17 a day	17¢ a mile (including gas)
Better Deals Rentals	$12 a day	22¢ a mile (including gas)
Cheap Car	$8 a day	10¢ a mile (not including gas)

Figure the gas at 14 MPG and 38¢ a gallon. Which car is the best deal?

What To Do Next

Quit out of BASIC, or go on to the next chapter.

Chapter 5. Testing Conditions

You might want a program to do different things depending on conditions that come up within it. To test those conditions, you can use an IF statement.

How the IF Statement Works

An IF statement can test whether a variable is equal to, greater than, or less than any other thing you choose to name. IF operates this way:

- First, the IF statement tests the condition you define.
- If the answer to the test is *yes, it's true*, BASIC will do what you direct it to do following the word THEN in the statement.
- If the answer is *no, it's not true*, the computer ignores the THEN part of the IF statement and does what you've directed following the word ELSE. If you don't have an ELSE, the computer goes directly to the next line in the program.

There are two kinds of IF statements in BASIC: **simple IF** and **IF block**. We'll show you sample programs with both.

Here's an example of a simple IF statement:

```
130 if x=0 then print 'X is zero'
```

Here, if X is equal to zero, BASIC prints `X is zero` on your screen. If X is not zero, it goes on to the next line.

Here's another example:

```
190 if age<18 then m$='minor' else m$='adult'
```

If the AGE variable is less than 18, the THEN part of the statement assigns the value "minor" to the M$ variable. Otherwise (that is, if the AGE variable is not less than 18), the ELSE part of the statement assigns "adult" to M4.

There are six tests you can make with IF. This is how you type them at the keyboard.

For This Test	Use This Key or Keys
equal to	= or EQ
not equal to	<> or >< or NE
greater than	> or GT
less than	< or LT
greater than or equal to	>= or => or GE
less than or equal to	<= or =< or LE

Some Examples

This IF statement	Means This
if x>=10 then let c=d else let c=0	If X is greater than or equal to 10, set variable C to value in D. Otherwise, let C equal 0.
If y LT 21 then print x	If Y is less than 21, print value of X on your screen.
If a1>=5 then print a1	If A1 is greater than or equal to 5, print A1.
If a2<>0 then print a2	If A2 is not equal to zero, print A2.

Program Example 1 — DISCOUNT Program

(Feel free to type in the following examples, if you like, or simply look at the programs and read the discussions. If you do type them in, remember to initialize before you begin, and save when you're finished.)

You are in charge of billing people for orders of dresses. There are two styles, one at $108 a dozen, and one at $136 a dozen. On orders of $500 or over, there is a 10% discount. For the account you are now working on, there are two dozen orders for the first dress, and three dozen orders for the second dress. Figure out the discounts on the orders.

The program illustrates the simple IF. With a simple IF, the word THEN and a following statement are on the same line as the IF.

Such lines might get to be too long to fit on the screen, but BASIC has a way to split long lines into pieces — the **continuation character (&)**. If you end a line with an ampersand, BASIC will automatically begin the next line with an ampersand (you should not type in a line number in the continued line), and will treat them like one continuous line. For example, line 150 in the DISCOUNT program below might look like this:

```
150 if t>=500 then let d=.10*t    &
  &           else let d=0
```

The program looks like this:

```
100 rem DISCOUNT program
110 rem Program to figure out discounts on orders
120 let a=2
130 let b=3
140 let t=a*108+b*136
150 if t>=500 then let d=.10*t else let d=0
160 print t,d,t-d
170 end
```

This program works like this:

1. Find the total order (line 140).
2. Test to see if it is at least $500 (line 150). If it is, figure out the discount and then continue to the next line and print out the total.

   Note: D will be zero when the order is less than $500, because each time the computer starts to execute a program, it automatically sets the value of all the numeric variables in the program to zero. Character variables are set to null.

(Null is an empty string: zero length, with nothing inside.) This is called **initializing values**. The values remain zeros or nulls until something in the program (like a LET statement) assigns a different value. This means that you never have any problems with values *being left over* from the last time you ran the program.

Program Example 2 — MOVING Program

You are moving. You get estimates from two movers and want to know which mover will be cheaper. Mover A charges $40 an hour and estimates the work will take 5 hours. Mover B charges $32.50 an hour and estimates the work will take 8 hours. If both movers cost the same, you'll take mover B because he has a better reputation.

The program illustrates the IF block statement. IF blocks are helpful when the program contains more than one statement to be executed when the condition is true or when the condition is false. The THEN clause is still on the line with the IF, but statements following it (including the ELSE statement) are on separate lines. The IF block must be ended by an END IF (END IF is always two words) statement. The ELSE statement is optional. Here is the program:

```
100 rem MOVING program
110 rem Calculates which mover is cheaper
120 let a=5*40
130 let b=8*32.50
140 rem Test to see who is cheaper
150 if a<b then ! a is cheaper
160   print "a is cheaper"
170   print a
180 else ! b is cheaper or same
190   print "b is cheaper or equal"
200   print b
210 end if
220 print "End of program MOVING"
230 end
```

Here is how this program works:

1. In lines 120 and 130, you figure the total cost for each mover.
2. In line 150, you make a test to determine one of two paths for your program. The basis of the test is which mover is cheaper.
3. If mover A is cheaper (that is, the statement a<b is *true*), lines 160 and 170 will be executed. The words `a is cheaper` (from line 160) and then the *value* of variable a (line 170) will be printed on your screen.
4. If mover B is cheaper (that is, the statement a<b is *false*), BASIC will branch to line 180 (the one beginning with ELSE). The words `b is cheaper or equal` (from line 190) and then the *value* of variable b (line 200) will be printed on your screen.
5. Line 210 ends the IF block, and line 230 ends the program. No matter which branch was taken (the "a is cheaper" or the "b is cheaper") the statement after the END IF will be executed. The words `End of program MOVING` will be printed before the program ends at line 230.

You can also nest IF statements. The MOVING2 program below illustrates nested IFs.

```
100 rem MOVING2 program
110 rem Calculates which mover is cheaper
120 let a=5*40
130 let b=8*32.50
140 rem Test to see who is cheaper
150 if a<b then        ! beginning of first if block
160   print "a is cheaper"
170   print a
180 else
190   if a>b then      ! beginning of second if block
200     print "b is cheaper"
210     print b
220   else
230     print "a and b are equally expensive"
240     print a
250   end if           ! end of second if block
260 end if             ! end of first if block
270 print "End of program MOVING2"
280 end
```

Summary

You learned the following statements in Chapter 5:

Statement	What It Does
ELSE	Marks the beginning of the ELSE block portion of an IF block.
END IF	Ends an IF block.
IF	Executes a logical expression and conditionally transfers control or conditionally executes a statement or series of statements.

Figure 6. Statements Presented in Chapter 5

Chapter 12, "Command and Statement Summary" on page 79 contains a complete summary of all statements and commands used in this book.

You also learned in this chapter about THEN and ELSE clauses.

Exercises

Write BASIC programs to solve one or more of the following problems. Name them as indicated, save them, and run them. The programs are in Appendix A, "Answers to Exercises" on page 81.

Problem 1 — CAPECOD Program

The Smiths and the Hoopers live next door to each other. They are both driving to Cape Cod, which is 350 miles away, for a vacation. The Smiths leave a 8 AM and average 48 miles an hour. The Hoopers leave at 10:30 AM and they drive at an average of 62 miles an hour. Print out the name of the couple who will get there first.

Problem 2 — APTMENT Program

You are looking for an apartment. You find two, one on your own, and one through an agent. You like them equally, so your choice will be based strictly on cost. Here are the details:

Apartment 1: The rent is $225 a month and does not include utilities. You figure $15 a month for gas and electric. There is a two-year lease.

Apartment 2: The rent is $230 a month. It includes gas and electricity. You must pay the agent 10% of a year's rent. The lease is for two years.

Which should you take? What will be your average monthly cost over the two years?

Problem 3 — TAX Program

Mr. and Mrs. Richards are figuring out their income tax, and they are trying to decide whether to file joint or separate returns. Mr. Richards' taxable income is $8750. Mrs. Richards' taxable income is $10,312.

For separate returns, this is the schedule:

```
taxable income $8000-10,000, pay $1630 + 28% of amount over $8000
taxable income $10,000-12,000, pay $2190 + 32% of amount over $10,000
```

For a joint return, this is the schedule:

```
taxable income $16,000-20,000, pay $3260 + 28% of amount over $16,000
```

How should they file? What would their taxes be for each way?

What To Do Next

Quit out of BASIC, or go on to the next chapter.

Chapter 6. Loops

Computers are especially handy for doing things *iteratively*, or repeating the same operations over and over again. Programming structures that allow iterative operations are called **loops**, which are explained in this chapter.

Here's a new problem. You are a rug salesman. All your rugs come in rolls 12 feet wide. Your customers buy rugs in varying lengths depending on how long their rooms are. You want to make up a chart of how many square yards of rug are required for rooms of different lengths. The most popular room sizes start at 9 feet long (a 12 by 9 foot rug) and go up a foot at a time (12 by 10, 12 by 11, and so on) until they reach 12 by 20.

Write a program that computes the number of square yards in a 12 by 9 rug, then a 12 by 10 rug, on up to a 12 by 20 rug.

To compute the number of square yards in each of the twelve different sized rugs, you have to find the number of square feet and divide by 9. The main computation step is:

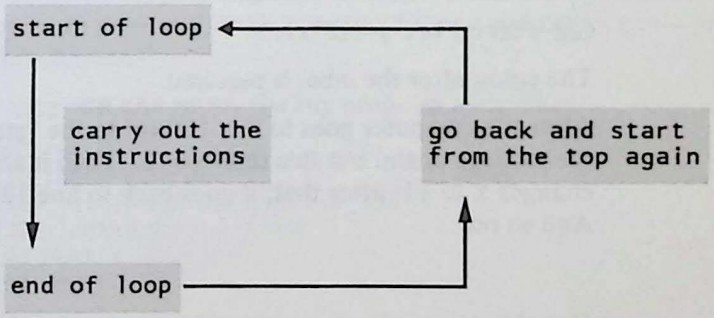
```
let y=(12*x)/9
```

Here Y is the number of square yards, and X is the length of each different rug. X would have to change from 9 to 10 on up to 20. You could write a program like this (*do not* type in this program):

```
100 let x=9
110 let y=(12*x)/9
120 print y
130 let x=10
140 let y=(12*x)/9
150 print y
```

and so on. But there must be a better way. There is — you can make a loop.

A loop is just what it sounds like. It is a series of program steps that go in a circle. It looks like this:



Two things have to happen to this loop to make it work.

1. It has to have some way to change the values it uses before it loops up to the top and starts again.
2. It has to have some way to know when to stop, or the program will run indefinitely.

A Loop Technique

So far you have these program instructions:

```
let y=(12*x)/9
print y
```

You want to start with X=9, so put a LET statement ahead of these two, assigning the value 9 as the first value of X. Now the program looks like this:

```
100 rem LOOP1 program
110 let x=9
120 let y=(12*x)/9
130 print y
```

To avoid specifying X=10, X=11, and so on, write a general statement that will just keep increasing X by 1. That statement is:

```
140 let x=x+1
```

Remember that while this statement looks peculiar in a mathematical sense, it's perfectly valid in BASIC.

By adding statement 140 to the program, you've changed the value of X and completed the steps required to make this loop operate once. Now you have to add a GOTO statement to make the computer go back to the beginning:

```
150 goto 120
```

A GOTO statement does just what it says: it directs the computer to *go to* the line number indicated. You go to line 120 because you only have to go back to the computation step, not to line 110 where you originally set X=9.

You can also direct BASIC to go to a **label** instead of a specific line number. That's really a better way to do it, because labels make programs easier to write now and to read later. You can use meaningful names up to 40 characters long.[16] Statement 150 would look like this with a label:

```
150 goto start
```

and line 120 would be changed to include the label "start":

```
120 start: let y=(12*x)/9
```

The colon after the label is required.

After the computer goes to line 120 (or to the "start" label) it figures and prints the yardage again, but this time X=10. Then it arrives back at line 140 and changes X to 11; after that, it goes back to line 120 to figure the next yardage. And so on.

Ending a Loop

One thing is missing from this otherwise perfect loop. It never ends. Not at X=20, not at X=30, because X just keeps going up and up. If you are sitting in front of your terminal while this program is running, you might be able to stop the loop by pressing ENTER. *If you run into real trouble and can't get a program in an infinite loop to stop, first try pressing the* **attention key** *(usually the key marked* PA1 *or* ATTN*). If that doesn't work, get your system administrator or your friend across the hall again.*

---

16 Programs with GOTO statements are often hard to read, and are not considered very elegant. We'll try to avoid GOTO's in our other examples. You should probably avoid them in your programs, as well.

You *can* make the loop stop automatically if you build in a test with an IF statement to see when you've processed enough Xs. In this case, enough is when the value of X passes 20.

This is the finished program (you can type this one in, if you like):

```
100 rem LOOP1 program
110 let x=9
120 start: let y=(12*x)/9
130 print y
140 let x=x+1
150 if x<=20 then goto start
160 end
```

Built-In Loops — FOR Statement

There is a better way to make a loop in a program. At the beginning of the loop, instead of setting X equal to its first value, you give the entire range of values that X will use. In the rug example, you would write:

```
100 for x=9 to 20
```

Then you write the instructions that solve the problem:

```
110 let y=(12*x)/9
120 print y
```

Then you tell the computer to go on to the next value of X and repeat the loop:

```
130 next x
```

FOR and NEXT statements always go in pairs: FOR at the beginning of the loop, and NEXT at the end. The computer automatically repeats the loop as many times as you told it to in the FOR statement. When it finishes, it goes on to the statement following the NEXT statement. Loops like this are called FOR loops, or FOR/NEXT loops.

BASIC also has DO loops: you instruct BASIC to *do* something *while* a certain condition is true, or *until* a certain condition is true. We'll concentrate on FOR/NEXT loops in this chapter, but we will show you an example of the same problem done with both FOR/NEXT loops and DO loops — problem 1 at the end of this chapter.[17]

Using FOR and NEXT, the rug program looks like:

```
100 rem LOOP2 program
110 for x=9 to 20
120   let y=(12*x)/9
130   print y
140 next x
150 end
```

(To make programs with loops more readable, the loops are often indented. We'll follow that convention in this book.)

---

17 For more information about DO loops, see *IBM BASIC Programming Guide, SC26-4027* or *IBM BASIC Language Reference, GC26-4026.*

In a FOR statement, you can name any numeric variable to be the changing variable, and you can make its range of values anything you want. The range doesn't even have to be given in constant numbers. You can use other variables for the range, for example:

```
160 for j=a to b
.
.
.
210 next j
```

The values of a and b will be used to determine the number of times the loop will be repeated.

Steps in Loops

When you write a FOR statement, the computer assumes you want the variable you name to change its value in steps of 1 (1 to 2 to 3, or 18 to 19 to 20 to 21, and so forth). However, sometimes you may want to skip over numbers and use just even numbers, or odd numbers, or every tenth number. If your loop requires something other than steps of 1, you can handle the situation whether you are using FOR and NEXT, or a LET statement to control the loop.

If you write a loop that uses a LET statement, you can write LET statements like these:

```
190 let x=x+2   ! to change x in steps of 2
310 let x=x+10  ! to change x in steps of 10
```

If you're using FOR and NEXT statements for an automatic loop, you add the word STEP and the size of the step to the FOR statement. Here are some examples:

```
100 for x=1 to 25 step 2
```

gives you odd values of x from 1 to 25 (1,3,5,7...).

```
120 for b=10 to 100 step 10
```

gives you 10, 20, 30, on up to 100. For even values of D from 1 to 20, you would write:

```
110 for d=2 to 20 step 2
```

Notice that D is set to 2 because the first even number you want is 2.

If you leave out the word STEP, you automatically get steps of 1.

Here's how you might use steps in a program: Suppose you wanted to figure estimates on rugs for very large rooms — but estimates that change in increments of five feet. You could change statement 110 in LOOP2 to:

```
110 for x = 10 to 200 step 5
```

Loop Problem — INTEGER Program

Try to figure out and type in the following program. If you have trouble, a solution appears below.

If you make a sum of integers like this:

1+2+3+4+5+6+7+8...

how many numbers can you add before you exceed 1000?

The Program:

```
100 rem INTEGER program
110 rem Sample loop program
120 let total=0
130 for num=0 to 1000
140    let total=total+num
150    if total>1000 then
160      print "You exceed 1000 when you try to add the number ";num
170      stop ! stops the program
180    end if
190 next num
200 end
```

Answer: You exceed 1000 when you try to add the number 45

Loops within Loops

Here's a problem where two things change.

Find the annual amount of interest (A) at interest rates I of 5%, 6%, 7%, 8%, 9%, and 10% on principals (P) ranging from $100 to $1000 in steps of $100.

This problem can be solved by a program that uses two loops — one for changing the interest and one for changing the principal. Do the interest loop first:

```
rem i-loop creates a loop for i to vary from 5% to 10%
for i=5 to 10 ! beginning of i-loop
  let a=(i/100)*p ! computes a
  print p,i,a ! prints a
next i ! end of i-loop
```

Now all that's left is to define P, since the program doesn't know where to get the values for P. The P loop has no computations of its own. It's only a definition of what P's values are. It goes like this:

```
for p=100 to 1000 step 100
  .
  .
  .
next p
```

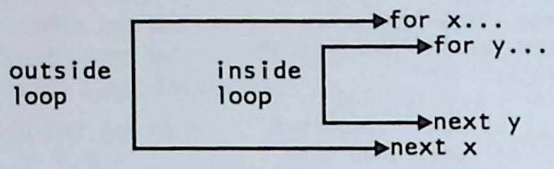
The P loop goes *around* the I loop.

Here's the whole program. Call it INTEREST and type it in.

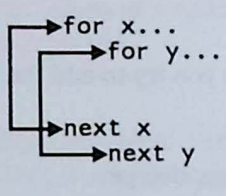
```
100 rem INTEREST program
110 rem Program to figure annual interest rates
120 for p=100 to 1000 step 100 ! beginning of p-loop
130   for i=5 to 10 ! beginning of i-loop
140     let a=(i/100)*p
150     print p,i,a
160    next i ! end of i-loop
170 next p ! end of p-loop
180 end
```

You put one loop entirely inside the other so that the computer will stay in one loop and finish it completely (*all* the values of I for a single P) before it goes on to the next P. In this program, the computer starts with P=100, then it comes to the I loop and sets I equal to 5. It goes on to compute the interest on $100 at 5%, 6%, 7%, 8%, 9%, and 10%, because it keeps repeating the I loop until I=10. When it finishes all the different interests on $100, it can go on to line 170, which is the bottom of the P loop. Here it increases P to $200, goes back to the top, and starts on the I loop again, this time with P=$200 and with I again ranging from 5 to 10. It continues like this until all the P's have been done. By then, you have all the amounts of interest you wanted.

Loops within loops must always be nested like this:



so that the inner loop is fully completed each time before the outside one is begun again. Two loops should never look like this:



One loop must always be completely enclosed by the other.

Here is another example. This is a program to find powers of X: X^2, X^3, X^4, and X^5 for X=1 to 10. Type it in. Call it POWERS.

Notice that the inside Y loop is fully contained in the outside X loop.

```
100 rem POWERS program
110 rem Calculates powers of X
120 for x=1 to 10 ! beginning of outside loop (change x from 1 to 10)
130   for y=2 to 5 ! beginning of inside loop (change y from 2 to 5)
140     print x;x**y
150   next y ! end of inside loop
160 next x ! end of outside loop
170 end
```

Summary

In this chapter, you learned the following new statements:

Statement	What It Does
DO	Initiates the execution of a set of statements that may be processed zero or more times.
FOR	Begins a FOR/NEXT count-condition loop.
GOTO	Branches to the indicated line number.
NEXT	Is the end delimiter of a FOR loop.
STOP	Terminates program execution.

Figure 7. Statements Presented in Chapter 6

Chapter 12, "Command and Statement Summary" on page 79 contains a complete summary of all statements and commands used in this book.

You also learned the STEP clause of the FOR statement.

Exercises

Write BASIC programs to solve one or more of the following problems. Name them as indicated, save them, and run them. The programs are in Appendix A, "Answers to Exercises" on page 81.

Problem 1 — POPULATE Program

In 1960, the population of Alpha City was 753,580. Its growth rate was 3.7% a year. In 1960, the population of Betaville was 529,430. Its growth rate was 5.1% a year. If the growth rates keep steady, when can you expect the population of Betaville to surpass that of Alpha City?

To solve the problem, create a FOR loop that counts years (that is, a statement that says `for 1961 to 2000`). Within the loop, figure the population of Alpha City (`a`) and Beta City (`b`), and print out the results when `b` becomes greater than `a`. Look again at the sum of integers program on page 46 if you need more help.

Problem 2 — SALESTAX Program

Make a sales-tax reference chart for yourself. The sales tax in your state is 7%, and you want your chart to go from $0 to $10 in steps of 10¢.

Problem 3 — CHANGE Program

A bank teller decides to do an experiment to liven up his day. Every time a customer asks for a change of a dollar, he makes change with a different combination of coins. Using half dollars, quarters, dimes, nickels, and pennies, how many customers can he give change to before he has to repeat a combination? Print out the combinations and the total number of combinations.

Note: This program causes about 12-15 screens of data to to be generated, as well as a final answer giving number of combinations possible.

What To Do Next

Quit out of BASIC, or go on to the next chapter.

Chapter 7. Better Ways to Put Values into Programs

In all the programs we've written, we've done this:

- Written a program to solve the problem using general expressions
- Supplied specific values for the expressions and run the program with the specific values

The advantage of programming in this way is that the bulk of the program doesn't change every time you want to solve the same problem with different numbers. You only need to change the numbers, not the programmed expression, when you want to run the program using different numbers.

We are now going to look at ways to supply specific numbers for programs. Once you know how to do this, you can keep your program expressions as general as you like.

More About the LET Statement

You already know the first way to supply values. You saw it briefly in Chapter 4 when we talked about generalizing the KETCHUP program. If you used LET statements throughout, KETCHUP might look like this:

```
100 rem KETCHUP program
110 let p1=39
120 let n1=12
130 let k1=p1/n1
140 let p2=55
150 let n2=15
160 let k2=p2/n2
170 let p3=47
180 let n3=14.5
190 let k3=p3/n3
200 print k1
210 print k2
220 print k3
230 end
```

Here, statements 110, 120, 140, 150, 170, and 180 supply actual values for the general formulas in the LET statements on lines 130, 160, and 190. If you want to change the values of P1, N1, P2, N2, P3, and N3 to find different unit prices, you would have to change the six statements on lines 110, 120, 140, 150, 170, and 180. (In this particular case, changing so many lines is not particularly convenient, but there are many cases where this LET statement technique is quite handy.)

The READ and DATA Statements

If there are a lot of numbers for you to supply in a program, writing a LET for each one can be tedious. Instead, you can use two statements called READ and DATA. You use these two statements together. This is what they do:

- READ tells the computer the names of all the variables you're going to supply values for.
- DATA supplies the values.

For the P1, N1, P2, N2, P3, and N3 of KETCHUP, READ looks like this:

```
130 read p1,n1,p2,n2,p3,n3
```

This means that BASIC will have to read values for variables P1, N1, P2, N2, P3, and N3.

The data statement looks like:

```
120 data 39,12,55,15,47,14.5
```

This gives the actual data (numbers, in this case) you're using for the program variables. Both READ and DATA use commas to keep the list of information in order. The values in the DATA statement must be listed in the same order that the variables are listed in the READ statement, so BASIC knows which variable gets which value.

You can use a single set of READ and DATA statements for *all* the variables in your program. You don't *have* to use only one READ/DATA set, though. You could also have one READ and several DATA statements, or vice versa.

If you wrote a program to compute interest at 5% on different amounts of money, you might want to use two sets of READ and DATA statements:

```
read i
data 5

read p
data 590
```

Then you could change the amounts of money without changing the 5% interest. You could also change the rate of interest without changing the amount of money.

If KETCHUP used READ and DATA statements, it would look like this:

```
100 rem KETCHUP program
110 rem Calculates unit prices for 3 brands of ketchup
120 data 39,12,55,15,47,14.5
130 read p1,n1,p2,n2,p3,n3
140 let k1=p1/n1
150 let k2=p2/n2
160 let k3=p3/n3
170 print k1
180 print k2
190 print k3
200 end
```

When you want to use other unit prices, you need only change line 120 of your program. By changing line 120, you can compute and print the unit price for any three items.

Note: You may find it easy to keep DATA and READ statements together as pairs. But you should know that there is no requirement for this. You can, if you want, put all the DATA statements together, at the beginning or end of your program, and scatter the READ statements throughout the program. You can do this because with READ and DATA, BASIC doesn't depend on placement in the program to find the right value for a variable. It depends only on the order in which you list the variables and the values. The first value encountered in a DATA statement is assigned to the first variable encountered in a READ statement, no matter where the two statements are in relation to each other; likewise, the tenth value encountered in a DATA statement is assigned to the tenth variable encountered in a READ statement, and so on. Only the order counts.

The INPUT Statement

Notice that the previous methods of giving values to variables have one thing in common:

The values are part of the program itself. You have to make a change to the actual program every time you want to change a value.

Another method for giving values to variables lets you type in the values when you execute your program. This means you can have saved a program, and any time you want to run it, days or weeks later, you can give the values to the computer on the spot, and not have to make changes to the program itself. It also means you

can write a general program before you know the specific numbers the program will work on.

To do this, you place an INPUT statement in your program. INPUT is just like READ, except it means that the values are going to come from the user at the terminal when the program is run instead of from a DATA statement within the program. Here's the INPUT statement for KETCHUP.

```
120 input prompt "Enter prices and weights": p1,n1,p2,n2,p3,n3
```

The prompt (the words inside the quotes) are a reminder to the user about what information is being requested. If you don't put in your own prompt in a program, BASIC will use a question mark. The colon between the prompt and p1 separates the prompt from the variable names for the data that will be entered.

Using this statement makes KETCHUP look like this:

```
100 rem UNITCOST program
110 rem Revision of KETCHUP program
120 input prompt "Enter prices and weights": p1,n1,p2,n2,p3,n3
130 let k1=p1/n1
140 let k2=p2/n2
150 let k3=p3/n3
160 print k1
170 print k2
180 print k3
190 end
```

If you enter, save, and run this version of KETCHUP (call the new program UNITCOST, since, in fact, it works for any other product as well), here's what happens:

```
run
Enter prices and weights _
```

Your response takes the place of a DATA statement. You type the values for P1, N1, P2, N2, P3, and N3, with commas between them, and press ENTER:

```
Enter prices and weights 39,12,55,15,47,14.5
3.25
3.66667
3.241379
 *
  _
```

After you press ENTER, the computer reads the numbers you entered, uses them in UNITCOST, and comes out with the answers. You can use any numbers you like with UNITCOST now. Try 6 ounces of a product at 79¢ versus 11 ounces at $1.53 versus 9.9 ounces at $1.29.

```
run
Enter prices and weights 79,6,153,11,129,9.9
13.1667
13.9091
13.0303
 *
  _
```

Run it again with your own values. They can be anything you like as long as they have commas between them, and as long as there are six values.

Getting Character Variables into Programs

We've been getting actual numbers into programs in this chapter, but any of the methods we've used will let you supply values for character variables as well. You have already seen how to do this with a LET statement. For INPUT and READ statements, you just use valid character variables where we've been using arithmetic variables.

For example, if you want a program to keep track of a person's height and weight, you can enter the person's name, height, and weight with READ and DATA like this:

```
100 read name$,height,weight
110 data Tom Jones, 6.1,184
```

You could also use an INPUT statement:

```
100 input prompt "Enter name,height,weight": name$,height,weight
```

and then respond to the prompt like this:

```
Enter name,height,weight Tom Jones,6.1,184
```

Summary

All these methods of giving values to variables are useful:

- LET statements
- READ and DATA statements
- INPUT statements with data supplied from the terminal

Just when you use each one will depend on what program you're running. You can use a combination of these methods if you have a program where some values don't change, some change occasionally, and others change often.

You learned the following new statements in this chapter:

Statement	What It Does
DATA	Creates internal data tables for reference by READ statements.
INPUT	Provides for input from the terminal.
READ	Retrieves the internal data tables created by DATA statements.

Figure 8. Statements Presented in Chapter 7

Chapter 12, "Command and Statement Summary" on page 79 contains a complete summary of all statements and commands used in this book.

Exercises

Problem 1 — BATTING2 Program

Write a general program for figuring out batting averages and slugging averages. When you run the program, you will have to enter the ballplayer's name (N$), the number of times he was up at bat (B), the number of home runs (R), triples (T), doubles (D), and singles (S). The program will compute the averages and will print the results. (The formulas are given in Problem 1 of the exercises for Chapter 4.)

Problem 2 — TYPING Program

You are giving typing tests to applicants for typing jobs. The test is five minutes long. After counting the number of words the applicant typed in five minutes, you subtract 10 words for each error, and then you divide the result by 5 for the final score. The minimum passing score is 50 words per minute, but if the score is between 45 and 49, you will let the applicant take the test one more time.

Write a general program that will let you type in the applicant's name, the total words, and the number of errors. The program should then compute the score and tell you if the applicant passed or not, or whether the applicant should be allowed to take the test again.

What To Do Next

Quit out of BASIC, or go on to the next chapter.

Chapter 8. More about Print

We've seen the PRINT statement used to print the value of variables, and to print back some material exactly as it was entered. And we've seen that to have material printed back exactly as you entered it, you put quotes around it. You should remember, then, that if you include this line in a program:

```
130 print 'my name is sam'
```

when the computer executes line 130, it types:

```
my name is sam
```

Similarly, if you type in:

```
140 print 'x'
```

then when line 140 is executed, you get back:

```
x
```

and not the *value* of X, which you would get if line 140 were:

```
140 print x
```

Within a single PRINT statement, you can mix quoted and unquoted material. You just use semicolons or commas as you do when there is more than one variable value to be printed. If you use semicolons, there will be no spaces between items unless they are numbers, in which case there will be one space between them. Commas will cause one item per **zone** to be printed — a zone is usually 20 characters wide. The semicolons or commas go after the end quote in quoted material.

Here's an example of a program named ANNUAL that computes annual interest for any rate and principal that you give it:

```
100 rem ANNUAL program
110 rem Calculates annual interest
120 input prompt "Enter rate and principal": r,p
130 let i=(r/100)*p
140 print 'Annual interest at';r; 'percent on $';p; 'is $';i
150 end
```

When you run this program and use 6% and $820, here's what you see:

```
Enter rate and principal 6,820
Annual interest at 6 percent on $ 820 is $ 49.2
```

Mathematical Calculations in PRINT Statements

PRINT allows you to include math calculations along with variables and words. Therefore, if you just want a calculation done and the result printed, you can do it in a single PRINT statement.

For example, you can write:

```
100 input x
110 print x,x**2
```

instead of writing:

```
100 input x
110 let y=x**2
120 print x,y
```

The arithmetic may be quite complicated, but as long as you separate each expression by semicolons or commas, PRINT can handle it.

Making Headings

Suppose you have a loop in a program named MILES that computes mileage allowances (at 12¢ a mile) for company auto trips (of 10 to 150 miles in steps of 5 miles).

```
100 rem MILES program
110 rem Computes mileage allowances
120 for x=10 to 150 step 5
130   print x,.12*x
140 next x
150 end
```

When you run this program, this is what you see at the terminal:

```
run
10      1.2
15      1.8
20      2.4
 .
 .
 .  (and so on)
```

You can make headings for these columns by putting another PRINT statement before the loop:

```
115 print 'MILES', 'MILEAGE ALLOWANCE'
```

Now the output looks like this:

```
MILES      MILEAGE ALLOWANCE
 10         1.2
 15         1.8
 20         2.4
 .          .
 .          .
 .          .
```

A Quick PRINT Statement Summary

Any time you want to print something exactly as you typed it, put it in quotes. You can mix quoted material with variables, and variables can include arithmetic operations. You use semicolons or commas between each separate item in a PRINT statement. Any spaces appearing in quoted material will appear when the line is printed.

Setting Up Your Own Format — PRINT USING and IMAGE

A more flexible way to set up your printed results is to use a statement called PRINT USING. This statement tells the computer to print variables, using a particular format. This format, however, is not part of the PRINT USING statement, as it was in the earlier examples of PRINT. You give the format in a separate program statement called an IMAGE statement. The PRINT USING statement is used together with an IMAGE statement.

The IMAGE statement is a full-fledged BASIC statement, but it looks different from most BASIC statements. The word IMAGE is followed by a colon (:). After the colon you type the exact wording and format that you want your printed results to have. You leave room for any variable values by using # signs where the values belong. When the program is run, the computer substitutes real values for the # signs, and prints the whole thing (including any blank spaces you leave). Think of IMAGE as a picture of what you want the output to look like, with the #s as placeholders. Here is a sample PRINT USING statement along with its IMAGE statement:

```
100 print using 110: g,n
110 image: GROSS SALES ARE ##### AND NET PROFITS ARE #####
```

Notice that you don't use quotes in line 110. This is true of all IMAGE statements, unless, of course, you want quotes to appear in your printed results.

If these two statements were part of a program, with G equal to 10732 and N equal to 2145, then at the time you ran the program, the computer would print:

```
GROSS SALES ARE 10732 AND NET PROFITS ARE  2145
```

When you put in the #'s, as stand-ins for the variable values, you are really telling BASIC how many spaces to leave for the values. If a value turns out to need more spaces than you left, you will get a row of asterisks (*) instead of the value when the computer prints the result. In these statements:

```
130 print using 140: a,a**3
140 image: A IS ## AND A CUBED IS ##
```

there are only two spaces for each value. If A is 3, the result is:

```
A IS  3 AND A CUBED IS 27
```

But if A is 5, then 5^3 is 125, which is three spaces long. The result is:

```
A IS  5 AND A CUBED IS **
```

The asterisks mean the answer didn't fit; it was too long for the space you left.

When you use # signs to leave room for your variable values in an IMAGE statement, you can also control how many decimal places you want the value to have when it is printed. You do this by inserting a decimal point in the string of #'s wherever you want the decimal point to go in the result. For example, when you are printing dollars and cents, insert a decimal point two places from the right, as in this example:

```
100 print using 110: s
110 image: THE SERVICE CHARGE FOR YOUR ACCOUNT IS $$$#.##
```

If S were 23.47, the result would look like this when you ran the program:

```
THE SERVICE CHARGE FOR YOUR ACCOUNT IS $23.47
```

The $ is used as a place holder (like the #), but with a difference — it "floats" to the front of the number (you'll get one $ right in front of your number). Try it with different values of s.

You can also use IMAGE with character variables. If you begin your string of # signs with <, the variables will be left justified; if you begin it with >, the variables will be right justified. If you use neither < nor > the variable will be centered. Example:

```
150 print using 160: name$, salary
160 image: <######### $####.##
```

If the image statement is short, you can include it with the print statement. Example:

```
150 print using "##.##": x
```

A PRINT Example

There are three sales people in your department, and you are responsible for making a monthly report showing their sales figures for each week of the month. You can write a program that will automatically print the report for you in an attractive format.

This is how the variables in the program are named. Each sales person is assigned a letter, A, B, and C. Starting with the first sales person:

A$ is the person's last name
A1 is the sales total for week 1
A2 is the sales total for week 2
A3 is the sales total for week 3
A4 is the sales total for week 4
A5 is the sales total for the month

There are other variables:

M$ is the name of the month.
W$, X$, Y$, Z$ represent the date of the last day of each week in the month.
T1, T2, T3, T4 are the totals for everybody for each week.
T5 is the total for everybody for the month.

For this program named REPORT, you have to supply the month's sales figures, the name of the month, and the last day of each week covered. The report is then printed automatically.

```
100 rem REPORT program
110 rem Program for printing monthly sales report
120 data 'Adler','Bipple','Cubbins'
130 read a$,b$,c$
140 input prompt'Enter name of month, last day of each week': m$,w$,x$,y$,z$
150 input prompt'Enter figures for a$': a1,a2,a3,a4
160 input prompt'Enter figures for b$': b1,b2,b3,b4
170 input prompt'Enter figures for c$': c1,c2,c3,c4
180 let a5=a1+a2+a3+a4
190 let b5=b1+b2+b3+b4
200 let c5=c1+c2+c3+c4
210 print using 220: m$
220 image: Monthly sales report for month of <########
230 print
240 print using 250
250 image: Sales Person          Week Ending                Total
260 print using 270: w$,x$,y$,z$
270 image:          ########  ########  ########  ########
280 print
290 print using 320: a$,a1,a2,a3,a4,a5
300 print using 320: b$,b1,b2,b3,b4,b5
310 print using 320: c$,c1,c2,c3,c4,c5
320 image: <######  #####.##  #####.##  #####.##  #####.##  #####.##
330 let t1=a1+b1+c1
340 let t2=a2+b2+c2
350 let t3=a3+b3+c3
360 let t4=a4+b4+c4
370 let t5=a5+b5+c5
380 print
390 print using 400: t1,t2,t3,t4,t5
400 image: Totals   #####.##  #####.##  #####.##  #####.##  #####.##
410 end
```

Note: The PRINT statement with no data following it (such as the one on line 230) merely prints a blank line.

Here is a sample execution of this program.

```
Enter name of month, last day of each week January,1/1/84,1/8/84,1/15/84,1/22/84
Enter figures for Adler 1245.70,759.65,846.00,2005.50
Enter figures for Bipple 694.69,738.57,1690.25,993.60
Enter figures for Cubbins 1038.63,1682.00,927.89,1694.00

Monthly sales report for month of  January

Salesman                    Week Ending                    Total
              1/1/84   1/8/84  1/15/84  1/22/84

Adler        1245.70   759.65   846.00  2005.50     4856.85
Bipple        694.69   739.57  1690.25   993.60     4118.11
Cubbins      1038.63  1682.00   927.89  1694.00     5342.52

Totals       2979.02  3180.22  3464.14  4693.10    14316.48
```

Summary

In this chapter you learned how to make headings in PRINT statements, and how to print a report with PRINT USING and IMAGE.

Statement	What It Does
IMAGE	Specifies the format of print lines.
PRINT USING	Displays data at your terminal according to the definition in the IMAGE statement referred to.

Figure 9. Statements Presented in Chapter 8

Chapter 12, "Command and Statement Summary" on page 79 contains a complete summary of all statements and commands used in this book.

Exercise — GRADES Program

You are a teacher and you want to write a program that will figure out and print a report of three midterm grades for each student, and a final exam. The program will be used by your students to calculate their own grades. Have the students enter their grades from the terminal. The final grade for the term will be a letter grade based on an average of the exam scores according to this table:

```
90 and over        A      (passing)
80-89              B
70-79              C
60-69              D
below 60           F      (not passing)
```

In computing the average, the final exam has twice as much weight as any of the other three exams. The printed report is meant for the students, not you or the office, so it should be intelligible to them, and it should be personalized. Tell the students whether they passed or not, and what their individual percentages and final grades were.

What To Do Next

Quit out of BASIC, or go on to the next chapter.

Chapter 9. Built-In Functions

In Chapter 4, we showed you how to use SRQ(X) to calculate the square root of X. When you type SQR(X), you are really asking for a calculation (taking the square root) to be done on variable X. This calculation is built right into the BASIC language so you don't have to compute square roots yourself. Built-in calculations are called **functions**. BASIC has many built-in functions to help make it easier for you to write programs.

Time and Date Functions

BASIC has four functions that automatically supply the time and date.

DATE	gives you year, number of days (numeric)
DATE$	gives you year/month/day (character)
TIME	gives you time in seconds (numeric)
TIME$	gives you the time in hours, minutes, and seconds (character)

You don't use any value, like the X in SQR(X), with these functions. You use these functions in your programs like you use any other variable. For example, if you have a program that prints a daily report of office activities, you can automatically include the date in a report heading like this:

```
100 print 'Daily activities for': date$
```

As another example, if you're keeping temperature data throughout the day, you can automatically associate the time and the temperature with statements like these:

```
100 input t1
110 print using 120: time$, t1
120 image: At ########### the temperature reading is ###
```

General Math Functions

You already know the SQR function. There are some other mathematical functions in BASIC that are useful.

ABS(X)	gives the absolute value of X
INT(X)	largest integer less than or equal to X
IP(X)	gives the integer part of X
MAX(X,Y,...)	finds the maximum value among the variables X, Y, ...
RND	generates a random number between 0 and 1
SGN(X)	determines the sign of variable X, and returns a value of −1, 0, or +1 depending on whether X is negative, zero, or positive

To use these functions, you just substitute the name of your own variable for the X inside the parentheses. You can also include expressions inside the parentheses, for example INT(X**2+Y*12). You might use SGN(X) to find out if X is positive. If you're using RND and want a new set of random numbers every time the program is run, put a RANDOMIZE statement before you use RND in a program statement.

Conversion Functions

BASIC has some built-in ways to convert from one measuring system to another. Here are four functions:

CEN(X)	gives the centigrade equivalent of X Fahrenheit degrees
FAH(X)	gives the Fahrenheit equivalent of X centigrade degrees
DEG(X)	gives the number of degrees in X radians
RAD(X)	gives the number of radians in X degrees

For example:

```
CEN(32)   gives you the value 0
FAH(0)    gives you the value 32
```

Other Functions

BASIC has functions that automatically perform trigonometric operations for you. Some examples are SIN(X) that gives the sine of X radians, and ASIN(X) that gives the arcsine (in radians) of X.

Also available are functions that automatically take logarithms and calculate exponentials.

BASIC also has some special functions to represent the values of π and epsilon and infinity. For example, if you want to use π in an equation, use the BASIC function PI. EPS is the smallest positive number BASIC can use. INF is the largest.

Getting HELP for Functions

You can get a complete list of the functions available to you in BASIC by typing the BASIC command:[18]

```
help function
```

Summary

In this chapter you learned some of the built-in functions available in BASIC, such as date and time functions, math functions, and conversion functions.

You learned the following statement:

Statement	What It Does
RANDOMIZE	Generates a new starting point for the list of pseudorandom numbers used by the RND function.

Figure 10. Statements Presented in Chapter 9

Exercise — GUESS Program

Write a program where the computer "thinks" of a number, and you have to guess what the number is. Use the RND function to generate the number.

What To Do Next

Quit out of BASIC, or go on to the next chapter.

---

18 You can get command and statement help, as well. Issue the command HELP BASIC to find out more about the BASIC help facility.

Chapter 10. Arrays

Suppose you are keeping statistics about the weather. You will probably have lots of data, like average temperatures, inches of rainfall, high temperatures, and so on. If you write a program to keep all this data on record, it would be a good idea to arrange the data in some orderly fashion; for example, you can keep the average temperature for each month together in one place, and the rainfall for each month together in another place.

With BASIC, you can keep groups of similar data together by organizing them into arrays. Arrays group data into categories; within each array are individual members that make up the actual data for that category.

In the weather example, suppose we were going to keep three sets of data:

- The names of the months
- The average temperature for each month
- The total rainfall for each month

We can look at this data in one of two ways:

- Three separate arrays like these lists:

ARRAY 1 Names of Months	ARRAY 2 Average Temperature	ARRAY 3 Rainfall
January	28	3.47
February	31	2.10
March	35	2.95
April	49	4.62
May	60	3.02
June	64	2.87
July	75	2.04
August	81	1.89
September	71	2.74
October	59	2.90
November	46	1.85
December	37	2.35

- One array, like this table:

Month	Temp	Rainfall
1	28	3.47
2	31	2.10
3	35	2.95
4	49	4.62
5	60	3.02
6	64	2.87
7	75	2.04
8	81	1.89
9	71	2.74
10	59	2.90
11	46	1.85
12	37	2.35

The table is really a combination of the three lists. Since you can read it in two directions, it is called a **two-dimensional array**. The lists are **one-dimensional arrays**.

It will be a lot easier to deal with the weather data if we keep it together in three one-dimensional arrays, than it would be if we considered it as 36 separate variables. This chapter will show you how to work with arrays in BASIC programs.

Defining an Array

When you want to work with an array, you must first tell BASIC that you're dealing with an array and not with ordinary variables. This is called **defining** your array.

Defining the array merely involves telling BASIC how big it is going to be (so BASIC can leave room for it), and telling BASIC what kind of data will be in it. (Later on, you fill in the data, but this is not part of defining the array.) The data for your arrays can be made up of numeric data or character data. You can define an array to have either kind of data in it, but it must have *only* that kind of data in it. You can't mix characters and numbers in a single array. That's why we used the numbers of the months instead of their names when we put the weather data into a two-dimensional array.

An array composed of numbers is called an **numeric array**. Naming conventions and restrictions are the same as those for numeric variables (see "A Note about Variable Names" on page 32).

An array composed of character data is called a **character array**. Naming conventions and restrictions are the same as those for character variables (see "A Note about Variable Names" on page 32).

To define either kind of array, you use a statement called DIM. In DIM, you name the array and put the size of it in parentheses after the name.

DIM for One-Dimensional Arrays

For a one-dimensional array, the size is a single number. Therefore, to define a numeric one-dimensional array A with 12 items in it, your DIM statement is:

```
100 dim a(12)
```

This statement tells BASIC that the largest subscript for A is 12. BASIC normally starts counting subscripts from zero. To get BASIC to start counting at 1, you need a statement:

```
option base 1
```

before your DIM statement.

To define character array N$ with 20 items in it, your statement is:

```
100 dim n$(20)
```

To define both in one statement:

```
100 dim a(12),n$(20)
```

DIM for Two-Dimensional Arrays

For two-dimensional arrays, the size is two numbers, one for each dimension. The first number is the number of rows in the array; the second number is the number of columns in the array.

To define array W with 12 rows and 3 columns, DIM looks like this:

```
100 dim w(12,3)
```

Character array A$, with 3 rows and 4 columns is defined by:

```
100 dim a$(3,4)
```

You can define all your arrays in a single DIM statement. You can also mix definitions of one- and two-dimensional arrays in a single DIM statement.[19]

---

19 BASIC can handle arrays of up to 7 dimentions.

Members of Arrays

Each individual item in an array is called a **member** or **element** of the array. When you want to refer to a particular member of an array, instead of to the whole array itself, you talk about the position of that member in the array. For example, if you want to refer to the third member of one-dimensional array H, you would refer to it as h(3). To refer to the member in the first row, third column of array W, you use w(1,3). The position goes in parentheses after the name of the array. For two-dimensional arrays, the first number is always the number of the row, the second number is always the number of the column.

If we look at the weather example as three one-dimensional arrays, we can call the array with the names of the months M$, the array of temperatures T, and the array of rainfall R. If we consider the example data as one two-dimensional array, called W, the numbers of the months are in column 1, the temperature data is in column 2, and the rainfall is in column 3. If you wanted to refer to January in a program statement, you would refer to either m$(1) or w(1,1).

Here are all the months and the way you refer to them in M$ and W.

This Month	Is in This Position	
	In array M$	In array W
January	m$(1)	w(1,1)
February	m$(2)	w(2,1)
March	m$(3)	w(3,1)
April	m$(4)	w(4,1)
May	m$(5)	w(5,1)
June	m$(6)	w(6,1)
July	m$(7)	w(7,1)
August	m$(8)	w(8,1)
September	m$(9)	w(9,1)
October	m$(10)	w(10,1)
November	m$(11)	w(11,1)
December	m$(12)	w(12,1)

If we include the temperature and rainfall data, the first element in each one-dimensional array — M(1), T(1), R(1) — or the first row in array W — W(1,1), W(1,2), W(1,3) — will be data for January; the second element in each one-dimensional array, or the second row in W, will be data for February; and so on.

So far, however, there is no data for any of the months in any of the arrays. We have only defined names and sizes. After you define an array, BASIC sets the values of *all* its members to zero (for numeric arrays) or null strings (for character arrays).

Assigning Values to Array Members

To assign values to array members (the names of the months, or the temperatures), you use the methods of assigning values that you've been using all along.

LET Statements

You can use a LET statement to assign a value to an individual member of an array. So if the average temperature for January is 28°, you could write either of these statements:

```
200 let t(1)=28
200 let w(1,2)=28
```

This is okay if you only have a few values to assign, but it will take forever if the array is large. In the weather example, we would need 36 separate LET statements to get all the data into the arrays. Nevertheless, LET is handy if you only want to assign a few values, or if you want to change a value you have already assigned.

Remember that if you are assigning a value to a member of a *character* array, you put the characters you are assigning in quotes. For example:

```
240 let m$(1)='January'
```

DATA and READ Statements

Another way to assign values is to use DATA and READ statements. You use these the same way you do for variables. For example:

```
200 read m$(1),m$(2),m$(3)
210 data 'January','February','March'
```

or

```
200 read w(1,1),w(2,1),w(3,1)
210 data 1,2,3
```

Again, when you are dealing with large amounts of data, listing them all separately in a READ statement is not practical. In this example, you can take advantage of a FOR/NEXT loop to help assign values:

```
200 for i=1 to 12
210   read t(i)
220 next i
230 data 28,31,35,49,60,64,75,81,71,59,46,37
```

or

```
200 for i=1 to 12
210   read w(i,2)
220 next i
230 data 28,31,35,49,60,64,75,81,71,59,46,37
```

These statements assign all the average temperature data to array T, or to the second column of array W. (In the case of array W, since we are *only* assigning values to the second column, we used a constant 2 in the READ statement.) You can't avoid specifying twelve values in the DATA statement, but a loop like this makes the READ statement easier to handle.

When dealing with array W, you could, in fact, use one READ statement and two loops to read *all* the data in at once. It would look like this:

```
200 for i=1 to 12
210   for j=1 to 3
220     read w(i,j)
230   next j
240 next i
```

Arranged this way, the loops let you enter the data for each row of the array in succession. Your DATA statements might look like this:

```
370 data 1,28,3.47
380 data 2,31,2.10
390 data 3,35,2.95
```

Here, we've put each row of array W in a separate DATA statement because it is easier to visualize the data that way. But in fact you could string out the data so that more than one row appears in a DATA statement, like this:

```
370 data 1,28,3.47,2,31,2.10,3,35,2.95
```

This way you could enter as much data with each DATA statement as will fit on a line. The important thing is that the data must appear in the same *order* as if you were typing it in row by row.

INPUT Statements

You can use INPUT statements to assign values to array members from your terminal. You can list all the member names in the INPUT statement, or else you can write a FOR/NEXT loop similar to the ones for READ to specify the names of the members that are to receive values.

For example, you can assign values to the one-dimensional rainfall array R like this:

```
120 input r(1),r(2),r(3),r(4),r(5)
```

or like this:

```
220 for i=1 to 12
230   input r(i)
240 next i
```

Or you can assign rainfall data to the third column of array W like this:

```
220 for i=1 to 12
230   input w(i,3)
240 next i
```

As with READ, you can write a double loop for an INPUT statement so that you can supply all the data for W at once. In all cases, you get a question mark when BASIC is ready for you to supply the data from the terminal. However, if your INPUT statement is in a loop, you get a question mark each time the loop is executed. That means you supply the next item of data, and so on. You will have to fill in the data just one item at a time, waiting for a question mark between each entry.

Another Way to Assign Values to Array Members

Instead of using a loop with READ or INPUT to assign values, you can label your array name with the letters MAT, and write a READ or INPUT statement like this:

```
200 read mat m$
210 input mat w
```

These statements tell BASIC to read in values for the *entire* array.

Here's the difference between working with array members individually and as members of an entire array:

- If you're working with array members individually, you write BASIC statements just as if you were working with ordinary variables.
- If you're working with an entire array, the BASIC statement will need the keyword MAT.

The letters MAT stand for the keyword "matrix." You can label the array name with MAT to mean that BASIC should expect values for an entire array.

This method of assigning values with a READ statement has no effect on your DATA statements. Therefore, to assign the temperature data to one-dimensional array T, you could write these statements:

```
200 read mat t
210 data 28,31,35,39,60,75,81,59,46,37
```

If you used a READ MAT W statement, you would have to enter the data for the whole array in DATA statements. You assign the data in a row-by-row order, like this:

```
200 read mat w
210 data 1,28,3.47
220 data 2,31,2.10
230 data 3,35,2.95
```

or like this:

```
200 read mat w
210 data 1,28,3,47,2,31,2,10,3,35,2,95
```

When you use READ, include in the DATA statement the exact number of items you need.

If you use an INPUT statement with the label MAT on the array name, BASIC will signal you with a ?, as usual, when it is ready for your data. If you are supplying values for a one-dimensional array, just type in all the values on a single line. If you are supplying values for a two-dimensional array, either enter all data on one line, or as much as you want on one line, ending with a comma and continuing on the next line. End any line you want to continue with a comma. The prompt from BASIC is always a ? after the first line. (The first line prompt depends on whether you've coded an input prompt or not.) Enter a / if you have no more data to enter, but haven't entered all the values prompted for.

Assigning Values to an Entire Array at Once

If you want every member of an array to have a single value, like all ones, or all zeros, you can do it with an assignment statement like this:

```
330 mat a=(0)
```

You could also assign every member of an array the value of a variable, or the value of an arithmetic expression, with this kind of statement:

```
350 mat t=(x)
```

or

```
360 mat m=(x+y*z)
```

The value you are assigning must go in parentheses, so that BASIC knows it is not an array.

Doing Calculations with Members of Arrays

After you assign values to members of your arrays, you can do calculations with individual members. You use members of arrays just as you use any BASIC variable in any BASIC statement. Nothing is different except that you are keeping a set of variables together for your own convenience in organizing data. Each member still has a value, and can act as an independent variable.

Printing Arrays

Members of arrays, like ordinary variables, can be used in any PRINT statement. Here are some examples:

```
310 option base 1
320 print t(3),t(4),M$(2),w(10,2),x,y,z
330 print 'the average rainfall for January is:',w(1,3)
340 print using 350: m$(3), r(3)
350 image: for the month of ######### the rainfall was #.##
```

In addition, you can print an entire array if you label the name of the array with MAT. For example, the statement:

```
390 print mat t
```

will print the entire one-dimensional temperature array T. Each member will be printed in a separate row. Each row of the table will be printed on a separate line with one blank line separating rows. Each element of the array will be put in a separate zone. The statement:

```
360 print mat w
```

will print the entire two-dimensional weather array W. It will be printed row by row.

Putting the One-Dimensional Weather Arrays Together in a Program

Now we'll put the three one-dimensional weather arrays, M$, T, and R, together into a sample program that will keep all the data on record and print it out when you run the program.

```
100 rem WEATHER1 program
110 rem This program keeps weather data
120 option base 1
130 dim m$(12),t(12),r(12)
140 for i=1 to 12
150   read m$(i),t(i),r(i)
160 next i
170 data January,   28, 3.47
180 data February,  31, 2.10
190 data March,     35, 2.95
200 data April,     49, 4.62
210 data May,       60, 3.02
220 data June,      64, 2.87
230 data July,      75, 2.04
240 data August,    81, 1.89
250 data September, 71, 2.74
260 data October,   59, 2.90
270 data November,  46, 1.85
280 data December,  37, 2.35
290 print 'Weather statistics for the year 1984'
300 print
310 print using 330
320 print using 340
330 image: month        average        inches of
340 image:              temperature    rainfall
350 print
360 for i=1 to 12
370   print using 390: m$(i),t(i),r(i)
```

```
380 next i
390 image: <#########     ###            #.##
400 end
```

This program uses FOR/NEXT loops to simplify handling the large number of values involved in these arrays. Notice that instead of writing a FOR/NEXT loop for each array when we were assigning values to the members, we wrote a single loop that worked across the three arrays instead of completing each 12-member array in turn. Of course, the DATA statements had to have their data in the same order.

We also used a loop to help with the printing. It let us use a single PRINT USING statement with a single IMAGE statement to print out twelve lines of data.

Using the Weather Array W in a Program

Now we'll do the same thing using the two-dimensional array W. This time we'll use READ MAT W and PRINT MAT W statements instead of using loops to read in the weather data and print it out at the end.

```
100 rem WEATHER2 program
110 rem This program keeps data in a 2-dim array
120 option base 1
130 dim w(12,3)
140 read mat w
150 data 1,28,3.47,2,31,2.10,3,35,2.95,4,49,4.62,5,60,3.02
160 data 6,64,2.87,7,75,2.04,8,81,1.89,9,71,2.74,10,59
170 data 2.90,11,46,1.85,12,37,2.35
180 print 'weather statistics for the year 1984'
190 print
200 print 'month','avg temp','rainfall'
210 print mat w
220 end
```

Arithmetic with Arrays

Suppose, instead of weather data for one year, you have weather data for two years. This data can be in two sets of arrays.

You are interested in averaging the temperatures and rainfall over the two years and making new arrays to contain the two-year averages.

To see how to do this, let's look at the two sets of temperature data. If you assume that they are in two one-dimensional arrays called A and B, then to find the average temperature for each month over the two years, you have to add the two temperatures for January and divide by two, add the temperatures for February and divide by two, and so on.

Addition and Subtraction

You can do all the addition in one step, adding the entire array A to the entire array B, with this statement:

```
310 mat c=a+b
```

Again, the letters MAT stand for "matrix." The statement causes each member of A to be added to the corresponding member of B and the result stored in the corresponding member of C. The same kind of addition statement works if you want to add two-dimensional arrays. If all the weather data for year 1 were in two-dimensional array T, and for year 2 in two-dimensional array U, and you wanted the result in V, the statement would be:

```
340 mat v=t+u
```

Each member of T is added to the corresponding member of U. This includes the columns with the numbers of the months and the columns with the rainfall.

Similarly, if you want to subtract each member of an array from the corresponding member of another array, you could write a statement like this:

```
350 mat c=a-b
```

The letters MAT always tell BASIC to work with an entire array.

Just remember that you should define all the arrays, including the one that is getting the results, in a DIM statement at the start of your program. Also, you can only add or subtract when all the arrays named in the operation are the same size and dimension as each other. You can't, for example, add a 14-member array to a 12-member array.

Multiplication and Division

Now what about dividing by two? Before we can do this, we must see how to multiply, because BASIC doesn't let you divide arrays directly, only multiply. You can multiply each member of an array (called A, for example) by a constant, a single variable, or an arithmetic expression with a statement like this:

```
330 mat c=(2)*a
```

The multiplier *always* goes in parentheses so BASIC knows it is not another array, and it must always go *before* the *. For division, you merely multiply the array by 1 over the divisor, or by a decimal number like .5. Therefore, to divide each member of array A by 2, you can write:

```
380 mat c=(.5)*a
```

Averaging Two Sets of One-Dimensional Arrays in a Program

If the weather data is kept in two sets of one-dimensional arrays, A and B for temperature, and C and D for rainfall, a program for averaging the two sets and assigning the results to master arrays T and R might look like this:

```
100 option base 1
110 dim m$(12),a(12),b(12),c(12),d(12),t(12),r(12)
120 read mat m$
130 data            [names of months]
140 read mat a, mat b, mat c, mat d
150 data
160 data            [data for arrays a, b, c, and d]
170 data
180 data
190 mat t=a+b
200 mat t=(.5)*t
210 mat r=c+d
220 mat r=(.5)*r
230 for i=1 to 12
240   print m$(i),t(i),r(i)
250 next i
260 end
```

Note that we defined T and R in the DIM statement on line 110, as well as M$, A, B, C, and D. Also note that we only need one array for names of the months, no matter how many years' worth of data we have stored in other arrays.

Averaging Two-Dimensional Arrays in a Program

If the two sets of weather data were stored in two-dimensional arrays X and Y, a program for averaging the data might look like this:

```
100 option base 1
110 dim x(12,3),y(12,3),w(12,3)
120 read mat x, mat y
130 data
140 data            [data for arrays x and y]
150 data
160 data
170 mat w=x+y
180 mat w=(.5)*w
190 print mat w
200 end
```

Note that we defined array W along with arrays X and Y in the DIM statement at the start of the program. It may also be interesting to note that the numbers of the months, which are in column 1 of both X and Y, are added in statement 170. But when we divide by 2 in statement 180, we get back the original numbers 1 through 12.

Summary

In this chapter you learned about defining arrays, working with members of arrays, printing arrays, doing arithmetic with arrays, and the MAT keyword. You also learned the following statement:

Statement	What It Does
DIM	Specifies the size of arrays.
OPTION BASE	Specifies whether subscripts for elements in an array begin at zero or one.

Figure 11. Statements Presented in Chapter 10

Exercise — TAXTABLE Program

Here is an Internal Revenue schedule for single taxpayers. Write a program using this schedule so that a clerk can enter any amount of taxable income and have the tax figured automatically.

Schedule X-Single Taxpayers Not Qualifying for Rates in Schedule Y or Z

If the amount on line 55 of form 1040 is over:	Column A But not over:	Column B pay:	Column C plus:	Column D of the amount over:
$ —	$ 500	$ 00	14%	$ 000
$ 500	$ 1000	$ 70	15%	$ 500
$ 1000	$ 1500	$ 145	16%	$ 1000
$ 1500	$ 2000	$ 225	17%	$ 1500
$ 2000	$ 4000	$ 310	19%	$ 2000
$ 4000	$ 6000	$ 690	21%	$ 4000
$ 6000	$ 8000	$ 1110	24%	$ 6000
$ 8000	$ 10,000	$ 1590	25%	$ 8000
$ 10,000	$ 12,000	$ 2090	27%	$ 10,000
$ 12,000	$ 14,000	$ 2630	29%	$ 12,000
$ 14,000	$ 16,000	$ 3210	31%	$ 14,000
$ 16,000	$ 18,000	$ 3830	34%	$ 16,000
$ 18,000	$ 20,000	$ 4510	38%	$ 18,000
$ 20,000	$ 22,000	$ 5230	40%	$ 20,000
$ 22,000	$ 26,000	$ 5990	45%	$ 22,000
$ 26,000	$ 32,000	$ 7590	50%	$ 26,000
$ 32,000	$ 38,000	$ 10,290	55%	$ 32,000
$ 38,000	$ 44,000	$ 13,290	60%	$ 38,000
$ 44,000	$ 50,000	$ 16,590	62%	$ 44,000
$ 50,000	$ 60,000	$ 20,190	64%	$ 50,000
$ 60,000	$ 70,000	$ 26,390	65%	$ 60,000
$ 70,000	$ 80,000	$ 32,790	66%	$ 70,000
$ 80,000	$ 90,000	$ 39,390	68%	$ 80,000
$ 90,000	$100,000	$ 46,190	69%	$ 90,000
$100,000	—	$ 53,090	70%	$100,000

To write this program, we are going to use four arrays:
A: maximum amount of income in each category
B: minimum tax for that bracket
C: percent at which excess is taxed
D: "amount over" column

We don't need to make an array for the minimum amount of income in each income bracket, because we will not use it; instead, we will test to see what bracket the income belongs to in the same way we tested the typing scores in the TYPING program in Chapter 7. Also, column D is exactly the same as column A except each element is "bumped" by one place.

The columns of the schedule are labeled to show which column corresponds to which array.

What To Do Next

Go on to Chapter 11.

Chapter 11. Where Do You Go from Here?

You've learned the fundamentals of programming with BASIC. At this point, you could go on to the *IBM BASIC Programming Guide* to learn about more advanced topics in IBM BASIC. In that book you'll learn (more) about:

- PF keys, including the handy "retrieve" key
- Character string manipulation
- Structured programming facilities
- Writing data to and reading data from files
- Different numeric data types
- Calling subprograms, including those written in other languages
- Compiling your programs to make them run faster
- Accessing data stored in relational data bases
- Displaying graphics

You might also want to look in *IBM BASIC/VM System Services* and *IBM BASIC/MVS System Services* for system-specific information, such as printing your programs on a hard-copy printer.

Other IBM BASIC Publications

Listed below are the titles and order numbers for other books in the IBM BASIC library you might like to read when you finish this book.

- *IBM BASIC General Information*, GC26-4023, gives you a general description of IBM BASIC.
- *IBM BASIC Programming Guide*, SC26-4027, covers more advanced topics in BASIC programming.
- *IBM BASIC Language Reference*, GC26-4026, lists and describes all the IBM BASIC statements and commands, and gives the complete rules for programming in BASIC.
- *IBM BASIC Language Reference Summary*, SX26-3736, provides you with a summary of the material covered in the language reference book.
- *IBM BASIC/VM System Services*, SC26-4028 and *IBM BASIC/MVS System Services*, SC26-4106, give you guidance and reference information on how to use the interfaces between IBM BASIC and your operating environment.

Chapter 12. Command and Statement Summary

Command	What It Does
AUTO	Numbers your program statements automatically.
BASIC	Allows you to connect with IBM BASIC.
DELETE	Removes statement lines from your workspace.
HELP	Displays information about BASIC.
INITIALIZE	Clears your workspace.
LIST	Displays program lines on the screen.
LOAD	Gets a program from a permanent file and puts it into your workspace.
PURGE	Removes files from your library.
QUERY	Permits interrogation of your user files.
QUIT	Ends your current IBM BASIC session.
RENUMBER	Renumbers statement lines in your program.
RUN	Directs BASIC to run your program.
SAVE	Puts a copy of your program into a permanent file.

Figure 12. Commands Presented in This Book

Statement	What It Does
!	Inserts remarks into your program.
DATA	Creates internal data tables for reference by READ statements.
DIM	Specifies the size of arrays, and the length of character variables and array elements.
DO	Initiates the execution of a set of statements that may be processed zero or more times.
ELSE	Marks the beginning of the ELSE block portion of an IF block.
END IF	Ends an IF block.
FOR	Begins a FOR/NEXT count-condition loop.
GOTO	Branches to the indicated line number.
IF	Executes a logical expression and conditionally transfers control or conditionally executes a statement or series of statements.

Figure 13 (Part 1 of 2). Statements Presented in This Book

Statement	What It Does
IMAGE	Specifies the format of print lines.
INPUT	Provides for input from the terminal.
LET	Assigns values to variables.
NEXT	Is the end delimiter of a FOR loop.
OPTION BASE	Specifies whether subscripts for elements in an array begin at zero or one.
PRINT	Displays data on your screen.
PRINT USING	Displays data at your terminal according to the definition in the IMAGE statement referred to.
RANDOMIZE	Generates a new starting point for the list of pseudorandom numbers used by the RND function.
READ	Retrieves the internal data tables created by DATA statements.
REM	Inserts remarks into your program.
STOP	Terminates program execution.

Figure 13 (Part 2 of 2). Statements Presented in This Book

Appendix A. Answers to Exercises

Chapter 2

Here's the checkbook balancing program (CHECKBAL):

```
100 rem CHECKBAL program
110 let a=97.50+15.00+68.75
120 let b=10.75+54.00+37.50
130 let c=a-b-.30
140 print c
150 end
```

Answer: 78.70 (78.7)

Chapter 3

Problem 1 (ALGEBRA):

```
100 rem ALGEBRA program
110 let y=7
120 let x=.3*y-1.5
130 print "x is"; x
140 end
```

Answer: .6

Problem 2 (LAWN):

```
100 rem LAWN program
110 rem Find the number of square feet
120 let f=70*80
130 rem Find the cost
140 let c=f*.115
150 print "The cost is"; c
160 end
```

Answer: C is 225.4

Problem 3 (HAMBURG):

```
100 rem HAMBURG program
110 rem How many hamburgers total?
120 let h=17*3+14*2
130 rem How many ounces is that?
140 let o=3*h
150 rem How many pounds?
160 let p=o/16
170 print p; "pounds of meat for" ;h; "hamburgers"
180 end
```

Answer: P is 14.8125; N is 79

Problem 4 (CHARTER):

```
100 rem CHARTER program
110 rem Find the base cost of each seat
120 let s= 10000/117
130 rem Add on the profit and the tax
140 let t=s+.05*s+2.50
150 print "Price of each ticket is"; t
160 end
```

Answer: 92.2436

Problem 5 (SOAP):

```
100 rem SOAP program
110 rem Figures the total number of washes in a gallon
120 let t=32*4*6
130 rem Divide by 60 washes a week to find number of weeks
140 let w=t/60
150 print w; "weeks"
160 rem Find how much hand cream needed in w weeks
170 let c=.5*7*w
180 print c; "ounces of hand cream"
190 end
```

Answer: W is 12.8; C is 448

Chapter 4

Problem 1 (BATTING1):

```
100 rem BATTING1 program
110 rem Find and print the batting average
120 let a=130/521
130 print a
140 rem Find and print the slugging average
150 let s1=15*4+6*3+21*2+(130-15-6-21)
160 let s=s1/521
170 print s
180 end
```

Answer: Batting average is .24952 and slugging average is .399232

Problem 2 (ELECTION):

```
100 rem ELECTION program
110 rem Election results
120 let a$='adams'
130 let a=4106+2890+2981
140 let b$='biller'
150 let b=3279+3515+2493
160 let c$='campbell'
170 let c=4002+3131+3012
180 let t=a+b+c
190 let a1=a/t
200 let b1=b/t
210 let c1=c/t
220 rem Print total vote
230 print t
240 print a$,a,a1
250 print b$,b,b1
260 print c$,c,c1
270 end
```

Answers: T (total) is 29409

```
Adams     9977   .33925
Biller    9287   .315788
Campbell 10145   .344962
```

Problem 3 (AIRCOND):

```
100 rem AIRCOND program
110 let a=5000/820
120 let b=6000/910
130 let c=5500/850
140 print a,b,c
150 end
```

Answers: a is 6.09756, b is 6.59341, and c is 6.47059

Problem 4 (RENTAL):

```
100 rem RENTAL program
100 let a$='acme'
110 let a=17+.17*275
120 print a$,a
130 let b$='better deals'
140 let b=12+.22*275
150 print b$,b
160 let c$='cheap car'
170 let c=8+.10*275+(275/14)*.38
180 print c$,c
190 end
```

Answers: a is 63.75, b is 72.50 (72.5), and c is 42.9643

Cheap (c) is cheapest

Chapter 5

Problem 1 (CAPECOD):

```
100 rem CAPECOD program
110 rem Who gets to Cape Cod first?
120 let smith = 350/48 - 2.5
130 let hooper = 350/62
140 if smith < hooper then
150   print "The Smiths get there first."
160 else
170   if hooper < smith then print "The Hoopers get there first."
180 end if
190 if hooper = smith then print "The Smiths and Hoopers arrive together."
200 end
```

Answer: The Smiths get there first

Problem 2 (APTMENT):

```
100 rem APTMENT program
110 rem Which apartment is cheaper?
120 let apt1=24*(225+15)
130 let cost1=apt1/24
140 let apt2=(24*230)+.10*(12*230)
150 let cost2=apt2/24
160 if apt1<=apt2 then
170   print "Apartment 1 is cheaper, cost is ",cost1
180 else
190   print "Apartment 2 is cheaper, cost is ",cost2
200 end if
210 end
```

Answer: Apartment 1 is cheaper, cost is 240

Problem 3 (TAX):

```
100 rem TAX program
110 rem Figure cost of separate income tax returns
120 let i1=8750
130 let i2=10312
140 let t1=1630+.28*(i1-8000)
150 let t2=2190+.32*(i2-10000)
160 let t3=t1+t2
170 rem Figure cost of joint return
180 let i3=i1+i2
190 let t4=3260+.28*(i3-16000)
200 rem See which is cheaper
210 if t3<t4 then
220   print "Cost of separate returns is less"
230 else
240   print "Cost of joint return is less"
250 end if
260 print "Joint return costs ";t4
270 print "Separate return costs ";t3
280 end
```

Answer: Cost of joint return is less; joint costs 4117.36 and separate costs 4129.84

Chapter 6

Problem 1 (POPULATE):

With a FOR/NEXT loop:

```
100 rem POPULATE program
110 rem Population growth program
120 let a=753580
130 let b=529430
140 for y=1961 to 2000
150   let a=a+.037*a
160   let b=b+.051*b
170   if b>a then
180   print "Betaville will have more people than Alpha City in";y
190   stop
200   end if
210 next y
220 end
```

Answer: Betaville will have more people than Alpha City in 1987

With a DO loop:

```
100 rem POPULATE program
110 rem Population growth program
120 let a=753580
130 let b=529430
140 let y=1961
150 do until b>a
160   let a=a+.037*a
170   let b=b+.051*b
180   let y=y+1
190 loop
200 print "Betaville will have more people than Alpha City in"; y-1
210 end
```

Note: y-1 is needed in line 200 because the program loops one number past the correct answer before the answer is printed.

Problem 2 (SALESTAX):

```
100 rem SALESTAX program
110 for x=0 to 10 step .10
120   print x, .07*x
130 next x
140 end
```

Problem 3 (CHANGE):

```
100 rem CHANGE program
110 rem Change for $1.00 program
120 let c=0
130 print "   Halves ";"Quarters    ";"Dimes  ";"Nickels  ";"Pennies  "
140 for h=0 to 2
150   for q=0 to (100-(50*h))/25
160     for d=0 to (100-(50*h)-(25*q))/10
170       for n=0 to (100-(50*h)-(25*q)-(10*d))/5
180         p=100-(50*h)-(25*q)-(10*d)-(5*n)
190         c=c+1
200         print using 210: h,q,d,n,p
210         image: ######## ######## ######## ######## ########
220       next n
230     next d
240   next q
250 next h
260 print "Number of ways is: "; c
270 end
```

Answer: Number of ways is 292

Chapter 7

Problem 1 (BATTING2):

```
100 rem BATTING2 program
110 rem General program to figure batting and slugging averages
120 input prompt "Ballplayer's name": n$
130 input prompt "Times at bat, runs, triples, doubles, singles ": b,r,t,d,s
140 let h=r+t+d+s
150 let a=h/b
160 print 'These are the statistics for: '; n$
170 print 'Total batting average is: '; a
180 let a2=(4*r+3*t+2*d+s)/b
190 print 'Slugging average is: '; a2
200 end
```

Problem 2 (TYPING):

```
100 rem TYPING program
110 rem Typing test program
120 print ! Print out blank line
130 input prompt "Enter name, number of words, number of mistakes: ": n$,w,m
140 let s=(w-10*m)/5
150 print "Test results for "; n$; ":"
160 if s>=50 then
170   print "Applicant passed test"
180 else
190   print ! print out blank line
200   print "Applicant did not pass test"
210   if s>=45 then
220     print "Has applicant taken test before? Enter 'yes' or 'no' "
230     input a$
240     if a$="no" then
250       print "Let applicant try again"
260     else
270       print "Applicant has had 2 tries"
280     end if
290   end if
300 end if
310 print ! print out blank line
320 end
```

Chapter 8

Here's the grading program (GRADES):

```
100 rem GRADES program
110 for i=1 to 30
120   print
130   print "Have all students been processed? Enter yes or no "
140   input a$
150   if a$='yes' then stop
160   print "Enter name of student"
170   input n$
180   print "Enter percentages for 3 midterms and final"
190   input e1,e2,e3,f ! exams and final
200   let avg=(e1+e2+e3+2*f)/5
210   let g$='A'
220   if avg<90 then g$='B'
230   if avg<80 then g$='C'
240   if avg<70 then g$='D'
250   if avg<60 then g$='F'
260   print "Final grades for: "; n$
270   print "Your grade on the first exam was: "; e1
280   print "Your grade on the second exam was: "; e2
290   print "Your grade on the third exam was: "; e3
300   print "Your grade on the final was: "; f
310   print "Your final average is ";avg;" and your final grade is ";g$
320   if avg>=60 then
330   print "Congratulations on passing this course!"
340   else
350   print "You did not pass. Please make an appointment to see the dean."
360   end if
370   next i
380 end
```

Chapter 9

Here's the number-guessing program (GUESS):

```
100 rem GUESS program
110 rem Program to play a number-guessing game
120 print "Guess the number I'm thinking of. It is between 1 and 100."
130 print "You have 8 tries."
140 randomize
150 let n=ip(rnd(0)*100+1)
160 for t=1 to 8
170   print "Make a guess"
180   input g
190   if g=n then print "Congratulations! You guessed it!": stop
200   if g>n then
210     print "My number is lower"
220   else
230     print "My number is higher"
240   end if
250 next t
260 if t >=8 then print "Sorry, you had 8 guesses. My number was"; n
270 end
```

Note: In line 150, RND(0) generated a random number between 0 and 1. Multiplying it by 100 scaled the number between 0 and 99. Since we wanted the number to be between 1 and 100, we added 1 to the result. Then we applied the IP function to extract just the integer part of the number.

Chapter 10

Here's the tax scheduling program (TAXTABLE):

```
100 rem TAXTABLE program
110 rem set up the arrays
120 option base 1
130 dim a(25),b(25),c(25),d(25)
140 read mat a
150 data 500,1000,1500,2000,4000,6000,8000,10000,12000,14000
160 data 16000,18000,20000,22000,26000,32000,38000,44000,50000
170 data 60000,70000,80000,90000,100000,1000000
180 read mat b
190 data 0,70,145,225,310,690,1110,1590,2090,2630,3210,3830
200 data 4510,5230,5990,7590,10290,13290,16590,20190,26390,32790
210 data 39390,46190,53090
220 read mat c
230 data 14,15,16,17,19,21,24,25,27,29,31,34,38,40
240 data 45,50,55,60,62,64,65,66,68,69,70
250 let d(1)=0
260 for i=1 to 24
270   let d(i+1)=a(i)
280 next i
290 rem get amount of taxable income
300 print 'Enter amount of taxable income; if finished, enter 0'
310 input x
320 if x=0 then goto 420
330 rem Test x to see which bracket it belongs to
340 for i=1 to 25
350   if x<=a(i) then goto 370
360 next i
370 rem Compute tax here
380 let t=b(i)+(c(i)/100)*(x-d(i))
390 print using 400: x, t
400 image: Tax on $####### is $######.##
410 goto 300
420 end
```

Index

Special Characters

A

B

C

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Numerics

B is for BASIC
An Introduction to IBM BASIC
GC26-4102-0

Reader's Comment Form

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

If you have applied any technical newsletters (TNLs) to this book, please list them here:

Last TNL _____

Previous TNL _____

Fold on two lines, tape, and mail. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.) Thank you for your cooperation.

GC26-4102-0

Reader's Comment Form

Fold and tape | Please do not staple | Fold and tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation
P.O. Box 50020
Programming Publishing
San Jose, California 95150

Fold and tape | Please do not staple | Fold and tape

IBM®

B is for BASIC An Introduction to IBM BASIC (File No. S370-40) Printed in U.S.A. GC26-4102-0

IBM

B is for BASIC
An Introduction to IBM BASIC

File Number S370-40

GC26-4102-00

Printed in U.S.A.